Gradual Program Analysis for Null Pointers

Anonymous author
Anonymous affiliation

Abstract

Static analysis tools typically address the problem of excessive false positives by requiring programmers to explicitly annotate their code. However, when faced with incomplete annotations, analysis tools are either too conservative, yielding false positives, or too optimistic, resulting in unsound analysis results. In order to flexibly and soundly deal with partially-annotated programs, we propose to build upon and adapt the gradual typing approach to abstract-interpretation-based program analyses. Specifically, we focus on null-pointer analysis and demonstrate that a gradual null-pointer analysis hits a sweet spot, by gracefully applying static analysis where possible and relying on dynamic checks where necessary for soundness. In addition to formalizing a gradual null-pointer analysis for a core imperative language, we build a prototype using the Infer static analysis framework, and present preliminary evidence that the gradual null-pointer analysis reduces false positives compared to two existing null-pointer checkers for Infer. Further, we discuss ways in which the gradualization approach used to derive the gradual analysis from its static counterpart can be extended to support more domains. This work thus provides a basis for future analysis tools that can smoothly navigate the tradeoff between human effort and run-time overhead to reduce the number of reported false positives.

2012 ACM Subject Classification Theory of computation → Abstraction

Keywords and phrases gradual typing, gradual verification, dataflow analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.23

Acknowledgements Anonymous acknowledgements

1 Introduction

Static analysis is useful [1], but underused in practice because of false positives [14]. A commonly-used way to reduce false positives is through programmer-provided annotations [4] that make programmers intent manifest. For example, Facebook's Infer Eradicate [10], Uber's NullAway [3], and the Java Nullness Checker from the Checker Framework [20] all rely on `@NonNull` and `@Nullable` annotations to statically find and report potential null-pointer exceptions in Java code. However, in practice, annotating code completely can be very costly [6]—or even impossible, for instance, when relying on third-party libraries and APIs. As a result, since non-null reference variables are used extensively in software [6], many tools assume missing annotations are `@NonNull`. But, the huge number of false positives produced by such an approach in practice is a serious burden. To address this pitfall, NullAway assumes that sinks (i.e. targets of assignments and bindings) are `@Nullable` and sources are `@NonNull`. Unfortunately, both strategies are unsound, and therefore programs deemed valid may still raise null pointer exceptions at run time.

This paper explores a novel approach to these issues by drawing on research in gradual typing [21, 22, 13] and its recent adaptation to gradual verification [2, 23]. We propose gradual program analysis as a principled, sound, and practical way to handle missing annotations. As a first step in the research agenda of gradual program analysis, this article studies the case of a simple null-pointer analysis. We present a general formal framework to derive gradual program analyses by transforming static analyses based on abstract interpretation [8]. Specifically, we study analyses that operate over first-order procedural imperative languages and support user-provided annotations. This setting matches the core language used by many

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:30
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

tools, such as Infer. In essence, a *gradual analysis* treats missing annotations optimistically, but injects run-time checks to preserve soundness. Crucially, the static portion of a gradual analysis uses the same algorithmic architecture as the underlying static analysis.[1]

Additionally, we ensure that any gradual analysis produced from our framework satisfies the *gradual guarantees*, adapted from Siek *et al.* [22] formulation for gradual typing. Any gradual analysis is also a *conservative extension* of the base static analysis: when all annotations are provided, the gradual analysis is equivalent to the base static analysis, and no run-time checks are inserted. Therefore, the gradual analysis smoothly trades off between static and dynamic checking, driven by the annotation effort developers are willing to invest.

To provide initial evidence of the applicability of gradual null-pointer analysis, we implement a gradual null-pointer analysis (GNPA) using Facebook's Infer analysis framework and report on preliminary experiments using the prototype.[2] The experiments show that a gradual null-pointer analysis can be effectively implemented, and used at scale to produce a reasonably small number of false positives in practice—fewer than Infer Eradicate as well as a more recent Infer checker, NullSafe.

The rest of the paper is organized as follows. In Section 2, we motivate gradual program analysis in the setting of null pointers by looking at how Eradicate, NullSafe, NullAway, and the Java Nullness Checker operate on example code with missing annotations, showcasing the concrete advantages of GNPA. Section 3 formalizes PICL, a core imperative language similar to that of Infer. Section 4 then presents the static null-pointer analysis (NPA) for PICL, which is then used as the starting point for the derivation of the gradual analysis. We describe our approach to gradualizing a static program analysis in Section 5, using GNPA as the running case study. Additionally, Section 5 includes a discussion of important gradual properties our analysis adheres to: *soundness*, *conservative extension*, and the *gradual guarantee*. All proofs can be found in the appendix submitted as anonymous supplementary material. We report on the preliminary empirical evaluation of an Infer GNPA checker called *Graduator* in Section 6. Section 7 discusses related work and Section 8 concludes. In the conclusion, we sketch ways in which the approach presented here could be applied to other analysis domains, highlight open venues for future work in the area of gradual program analysis.

2 Gradual Null-Pointer Analysis in Action

This section informally introduces gradual null-pointer analysis and its potential compared to existing approaches through a simple example. We first briefly recall the basics of null-pointer analyses, and then discuss how current tools deal with missing annotations in problematic ways.

2.1 Null-Pointer Analysis in a Nutshell

With programming languages that allow any type to be inhabited by a null value, programmers end up facing runtime errors (or worse if the language is unsafe) related to dereferencing null pointers. A null-pointer analysis is a static analysis that detects *potential* null pointer dereferences and reports them as warnings, so that programmers can understand where

---

[1] Note that an alternative is phrasing nullness as a type system, which can also be gradualized [5, 18]. We focus on approaches based on static analysis, which have very different technical foundations and user experience. We compare to type-based approaches in Section 7.

[2] The Infer GNPA implementation and experiments will be submitted as an artifact.

explicit nullness checks should be added in order to avoid runtime errors. Examples of null-pointer analyses are Infer Eradicate [11] and the Java Null Checker [20]. Typically, a null-pointer analysis allows programmers to add annotations in the code to denote which variables (as well as fields, return values, etc.) are, or should be, non-null–e.g. `@NonNull`–and which are potentially null–e.g. `@Nullable`. A simple flow analysis is then able to detect and report conflicts, such as when a nullable variable is assigned to a non-null variable.

While a static null pointer analysis brings guarantees of robustness to a codebase, its adoption is not necessarily seamless. If a static analysis aims to be sound, it must not suffer from false negatives, i.e. miss any actual null pointer dereference that can happen at runtime. While desirable, this means the analysis necessarily has to be conservative and therefore reports false positives—locations that are thought to potentially trigger null pointer dereferences, but actually do not.

This standard static analysis conundrum is exacerbated when considering programs where not all variables are annotated. Of course, in practice, a codebase is rarely fully annotated. Existing null-pointer analyses assign missing annotations a concrete annotation, such as `Nullable` or `NonNull`. In doing so, they either report additional false positives, suffer from false negatives (and hence are unsound), or both. The rest of this section illustrates these issues with a simple example, and discusses how a gradual null-pointer analysis (GNPA) alleviates them. GNPA treats missing annotations in a special manner, following the gradual typing motto of being optimistic statically and relying on runtime checks for soundness [21]. Doing so allows the analysis to leverage both static and dynamic information to reduce false positives while maintaining soundness.

2.2 Avoiding False Positives

GNPA can reduce the number of false positives reported by static tools by leveraging provided annotations and run-time checks. We demonstrate this with the unannotated program in Figure 1. The program appends the reverse of a non-null string to the reverse of a null string and prints the result. The `reverse` method (lines 3–8) returns the reverse of an input string when it is non-null and an empty string when the input is `null`. Additionally, `reverse` is unannotated, as highlighted for reference.

The most straightforward approach to handling the missing annotations is to replace them with a fixed annotation. Infer Eradicate and the Java Nullness Checker both choose `@NonNull` as the default, since that is the most frequent annotation used in practice [6]. Thus, in this example, they would treat `reverse`'s argument and return value as annotated with `@NonNull`. This correctly assigns `reversed` and `frown` as non-null on lines 11 and 12; and consequently, no false positive is reported when `reversed` is dereferenced on line 13. However, both tools will report a false positive each time `reverse` is called with `null`, as in line 11.

Other uniform defaults are possible, but likewise lead to false positive warnings. For example, choosing `@Nullable` by default would result in a false positive when `reversed` is dereferenced. A more sophisticated choice would be the Java Nullness Checker's `@PolyNull` annotation, which supports type qualifier polymorphism for methods annotated with `@PolyNull`. If `reverse`'s method signature is annotated with `@PolyNull`, then `reverse` would have two conceptual versions:

```
static @Nullable String reverse(@Nullable String str)
static @NonNull String reverse(@NonNull String str)
```

At a call site, the most precise applicable signature would be chosen; so, calling `reverse` with

```
1  class Main {
2
3    static      String reverse(      String str) {
4      if (str == null) return new String();
5      StringBuilder builder = new StringBuilder(str);
6      builder.reverse();
7      return builder.toString();
8    }
9
10   public static void main(String[] args) {
11     String reversed = reverse(null);
12     String frown = reverse(":)");
13     String both = reversed.concat(frown);
14     System.out.println(both);
15   }
16 }
```

Figure 1 Unannotated Java code safely reversing nullable strings.

`null` (line 11) would result in the `@Nullable` signature, and calling `reverse` with `":)"` (line 12) would result in the `@NonNull` signature. Unfortunately, this strategy marks `reversed` on line 11 as `@Nullable` even though it is `@NonNull`, and a false positive is reported when `reversed` is dereferenced on line 13. So while `@PolyNull` increases the expressiveness of the annotation system, it does not solve the problem of avoiding false positives from uniform annotation defaults.

In contrast, GNPA optimistically assumes both calls to `reverse` in `main` (lines 11–12) are valid without assigning fixed annotations to `reverse`'s argument or return value. Then, the analysis can continue relying on *contextual optimism* when reasoning about the rest of `main`: `reversed` is assumed `@NonNull` to satisfy its dereference on line 13. Of course this is generally an unsound assumption, so a run-time check is inserted to ascertain the non-nullness of `reversed` and preserve soundness. Alternatively, a developer could annotate the return value of `reverse` with `@NonNull`. GNPA will operate as before except it will leverage this new information during static reasoning. Therefore, `reversed` will be marked `@NonNull` on line 11 and the dereference of `reversed` on line 13 will be statically proven safe without any run-time check.

It turns out that a non-uniform choice of defaults can be optimistic in the same sense as GNPA. For example, NullAway assumes sinks are `@Nullable` and sources are `@NonNull` when annotations are missing. In fact, this strategy correctly annotates `reverse`, and so no false positives are reported by the tool for the program in Figure 1. However, in contrast to the gradual approach, the NullAway approach is in fact unsound, as illustrated next.

2.3 Avoiding False Negatives

When Eradicate, NullAway, and the Java Nullness Checker handle missing annotations, they all give up soundness in an attempt to limit the number of false positives produced.

To illustrate, consider the same program from Figure 1, with one single change: the `reverse` method now returns `null` instead of an empty string (line 4).

```
    if (str == null) return null;
```

All of the tools mentioned earlier, including NullAway, erroneously assume that the return value of `reverse` is `@NonNull`. On line 11, `reversed` is assigned `reverse(null)`'s return value of `null`; so, it is an error to dereference `reversed` on line 13. Unfortunately, all of the tools assume `reversed` is assigned a non-null value and do not report an error on line 13. This is a *false negative*, which means that at runtime the program will fail with a null-pointer exception.

GNPA is similarly optimistic about `reversed` being non-null on line 13. However, GNPA safeguards its optimistic static assumptions with run-time checks. Therefore, the analysis will correctly report an error on line 13. Alternatively, a developer could annotate the return value of `reverse` with `@Nullable`. By doing so, the gradual analysis will be able to exploit this information statically to report a static error, instead of a dynamic error.

To sum up, a gradual null-pointer analysis can reduce false positives by optimistically treating missing annotations, and preserve soundness by detecting errors at runtime. Of course, one may wonder why it is better to fail at runtime when passing a null value as a non-null annotated argument, instead of just relying on the upcoming null-pointer exception. There are two answers to this question. First, in unsafe languages like C, a null-pointer dereference results in a crash. Second, in a safe language like Java where a null-pointer dereference is anyway detected and reported, it can be preferable to fail as soon as possible, in order to avoid performing computation (and side effects) under an incorrect assumption. This is similar to how the eager reporting of gradual typing can be seen as an improvement over simply relying on the underlying safety checks of a dynamically-typed language.

Next, we formally develop GNPA, prove that it is sound, and prove that it smoothly trades-off between static and dynamic checking following the gradual guarantee criterion from gradual typing [22]. We finally report on an actual implementation of GNPA and compare its effectiveness with existing tools.

3 PICL: A Procedural Imperative Core Language

Following the Abstract Gradual Typing methodology introduced by Garcia *et al.* [13], we build GNPA on top of a static null-pointer analysis, NPA. Thus, we first formally present a procedural imperative core language (PICL), used for both analyses to operate on; we present NPA in Section 4, and GNPA in Section 5. PICL is akin to the intermediate language of the Infer framework, and therefore the formal development around PICL drove the implementation of the Infer GNPA checker we evaluate in Section 6.

3.1 Syntax & Static Semantics

The syntax of PICL can be found in Figure 2. Programs consist of procedures[3], fields, and statements. Statements include the empty statement, sequences, variable declarations, variable and field assignments, conditionals, while loops, and returns. Expressions consist of

[3] Procedures accept only one parameter to simplify later formalisms.

$$
\begin{array}{rcl}
x, y & \in & \textsc{Var} \\
e & \in & \textsc{Expr} \\
a & \in & \textsc{Ann} = \{\texttt{Nullable},\ \texttt{NonNull},\ \texttt{?}\} \\
P & ::= & \overline{procedure}\ \overline{field}\ s \\
field & ::= & T\ f; \\
procedure & ::= & T\texttt{@}a\ m\ (\ \overline{T\texttt{@}a\ x}\ )\ \{\ s\ \} \\
T & ::= & \texttt{ref} \\
\oplus & ::= & \wedge \mid \vee
\end{array}
\qquad
\begin{array}{rcl}
m & \in & \textsc{Proc} \\
f & \in & \textsc{Field} \\
s & \in & \textsc{Stmt} \\
e & ::= & \texttt{null} \mid x \mid e \oplus e \mid e.f \mid \texttt{new}(\overline{f}) \\
 & & \mid m(x) \\
c & ::= & e = \texttt{null} \mid e \neq \texttt{null} \\
s & ::= & \texttt{skip} \mid s\ ;\ s \mid T\ x \mid x\ \texttt{:=}\ e \\
 & & \mid x.f\ \texttt{:=}\ y \mid \texttt{if}\ (c)\ \{\ s\ \}\ \texttt{else}\ \{\ s\ \} \\
 & & \mid \texttt{while}\ (c)\ \{\ s\ \} \mid \texttt{return}\ y
\end{array}
$$

Figure 2 Abstract syntax of PICL.

$$
\begin{array}{rcl}
x, y, z & \in & \textsc{Var} \\
a, b & \in & \textsc{Ann} = \{\texttt{Nullable},\ \texttt{NonNull},\ \texttt{?}\}
\end{array}
\qquad
\begin{array}{rcl}
m & \in & \textsc{Proc} \\
f & \in & \textsc{Field}
\end{array}
$$

$$
\begin{array}{rl}
I ::= & x\ \texttt{:=}\ y \mid x\ \texttt{:=}\ \texttt{null} \mid x\ \texttt{:=}\ m\texttt{@}a(y\texttt{@}b) \mid x\ \texttt{:=}\ \texttt{new}(\overline{f}) \mid x\ \texttt{:=}\ y \wedge z \mid x\ \texttt{:=}\ y \vee z \\
 & \mid x\ \texttt{:=}\ y.f \mid x.f\ \texttt{:=}\ y \mid \texttt{branch}\ x \mid \texttt{if}\ x \mid \texttt{else}\ x \mid \texttt{return}\ y\texttt{@}a \mid \texttt{main} \\
 & \mid \texttt{proc}\ m\texttt{@}a(y\texttt{@}b)
\end{array}
$$

Figure 3 Abstract syntax of a CFG instruction.

`null` literals, variables, comparisons, conjunctions, disjunctions, field accesses, object allocations, and procedure calls. Finally, procedures may have `Nullable` or `NonNull` annotations on their arguments and return values. Missing annotations are represented by `?`.

As the focus of this work is not on typing, we only consider well-formed and well-typed programs, which is standard and not formalized here. In particular, variables are declared and initialized before use, and field and procedure names are unique.

3.2 Control Flow Graph Representation

Well-formed programs written in the abstract syntax given in Fig. 2 are translated into *control flow graphs*—one graph for each procedure body and one for the main s. A finite control flow graph (CFG) for program p has vertices $\textsc{Vert}_p$ and edges $\textsc{Edge}_p \subseteq \textsc{Vert}_p \times \textsc{Vert}_p$. For $v_1, v_2 \in \textsc{Vert}_p$, we write $v_1 \xrightarrow{p} v_2$ to denote $(v_1, v_2) \in \textsc{Edge}_p$. Each vertex holds a single instruction, which we can access using the function $\textsc{inst}_p : \textsc{Vert}_p \to \textsc{Inst}$. We write $[\iota]_v$ to denote a vertex $v \in \textsc{Vert}_p$ such that $\textsc{inst}_p(v) = \iota$, or just $[\iota]$ (omitting the v) when the vertex itself is not important. By construction, these translated CFGs satisfy certain well-formedness properties, listed in the appendix.

The set of possible instructions is defined in Figure 3. In general, the CFG instructions are atomic variants of program statements designed to simplify the analysis presentations. Figure 4 gives the CFG of a simple procedure `foo`, which calls `bar` repeatedly until `x` becomes non-null and then returns `x`. The CFG starts with `foo`'s entry node `proc` $foo@NonNull(x@Nullable)$ (similarly, `main` is always the entry node of the main program's CFG). Then, the while loop on lines 3–6 results in the `branch` x sub-graph, which leads to `if` x when x is non-null and `else` x when x is null. The call to `bar` follows from `else` x and loops back to `branch` x as expected. Finally, `return` $x@NonNull$ follows from `if` x ending the CFG. Precise semantics for instructions is given in Section 3.3.

```
ref@NonNull foo(ref@Nullable x)
{
  while (x == null)
  {
    x := bar(x);
  }
  return x;
}
```

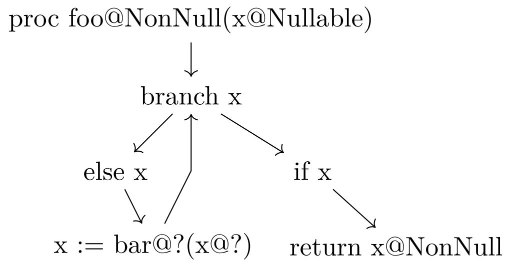

Figure 4 Example CFG.

3.3 Dynamic Semantics

We define the set of possible object locations as the set of natural numbers and 0, $\textsc{Val} = \mathbb{N} \cup \{0\}$. The `null` pointer is location 0.

Now, a program state ($\textsc{State}_p \subseteq \textsc{Stack}_p \times \textsc{Mem}_p$) consists of a stack and a heap. A heap $\mu \in \textsc{Mem}_p = (\textsc{Val} \setminus \{0\}) \rightharpoonup (\textsc{Field} \rightharpoonup \textsc{Val})$ maps object locations and field names to program values—other (possibly null) pointers. A stack is made of stack frames each containing a local variable environment and CFG node:

$$S \in \textsc{Stack}_p ::= E \cdot S \mid \mathsf{nil} \quad \text{where} \quad E \in \textsc{Frame}_p = \textsc{Env} \times \textsc{Vert}_p$$
$$\text{and} \quad \textsc{Env} = \textsc{Var} \rightharpoonup \textsc{Val}.$$

Further, we restrict the set of states $\xi = \langle \langle \rho_1, v_1 \rangle \cdot \langle \rho_2, v_2 \rangle \cdots \langle \rho_n, v_n \rangle \cdot \mathsf{nil} \parallel \mu \rangle \in \textsc{State}_p$ to include only those satisfying the following conditions:

1. *Bottom stack frame is in* ***`main`****:* Let $\textsc{descend} : \textsc{Vert}_p \to \mathcal{P}^+(\textsc{Vert}_p)$ give the descendants of each node in the control flow graph. Then $v_i \in \textsc{descend}(v_0)$ if and only if $i = n$.
2. *Every variable defaults to* ***`null`*** *(except on* ***`main`*** *and* ***`proc`*** *nodes):* If $\textsc{inst}_p(v_i) \neq$ `main` and $\textsc{inst}_p(v_i) \neq$ `proc` $m@a(y@b)$ then ρ_i is a total function.
3. *Follow the "true" branch when non-null:* If $\textsc{inst}_p(v_i) =$ `if` y then $\rho_i(y) \neq 0$.
4. *Follow the "false" branch when null:* If $\textsc{inst}_p(v_i) =$ `else` y then $\rho_i(y) = 0$.
5. *Every frame except the top is a procedure call:* If $v_i \in \textsc{descend}($`proc` $m@a(y@b))$ then $\textsc{inst}_p(v_{i+1}) = x$ `:=` $m@a(y'@b)$, and either $b = ?$ or $\rho_{i+1}(y') \in \textsc{conc}(b)$ (see section 4.

Now, the small-step semantics of PICL is given in Figure 5, where $\rho_0 = \{x \mapsto 0 : x \in \textsc{Var}\}$. The rules rely on the following helper functions:

$$\begin{aligned}
\textsc{new} &: \textsc{Mem}_p \to \textsc{Val} \setminus \{0\} & \textsc{new}(\mu) &= 1 + \max(\{0\} \cup \mathrm{dom}(\mu)) \\
\textsc{branch} &: \textsc{Val} \times \textsc{Var} \to \textsc{Inst} & \textsc{branch}(n, x) &= \texttt{if}\ x \text{ if } n > 0;\ \texttt{else}\ x \text{ otherwise} \\
\textsc{and} &: \textsc{Val} \times \textsc{Val} \to \textsc{Val} & \textsc{and}(n_1, n_2) &= n_2 \text{ if } n_1 > 0;\ n_1 \text{ otherwise} \\
\textsc{or} &: \textsc{Val} \times \textsc{Val} \to \textsc{Val} & \textsc{or}(n_1, n_2) &= n_1 \text{ if } n_1 > 0;\ n_2 \text{ otherwise}
\end{aligned}$$

Notably, `branch` y steps to the `if` y node when y is non-null and `else` y when y is null. Additionally, if a procedure call's argument disagrees with its parameter annotation, then it will get stuck (rule 5 for states); otherwise, the call statement will safely step to the procedure's body. In contrast, the semantics will get stuck if a return value does not agree with the procedure's return annotation.

$$
\begin{aligned}
&\langle\langle \rho, [x \texttt{ := } y]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto \rho(y)], v\rangle \cdot S \parallel \mu\rangle\\
&\langle\langle \rho, [\texttt{branch } y]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho, [\textsc{branch}(\rho(y), y)]_v\rangle \cdot S \parallel \mu\rangle\\
&\langle\langle \rho, [\texttt{if } y]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho, v\rangle \cdot S \parallel \mu\rangle\\
&\langle\langle \rho, [\texttt{else } y]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho, v\rangle \cdot S \parallel \mu\rangle\\
&\langle\langle \rho, [x \texttt{ := } m\texttt{@}a(y\texttt{@}b)]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \varnothing, [\texttt{proc } m\texttt{@}a(y'\texttt{@}b)]\rangle \cdot \langle \rho, u\rangle \cdot S \parallel \mu\rangle\\
&\langle\langle \rho_1, [\texttt{proc } m\texttt{@}a(y\texttt{@}b)]_u\rangle \cdot \langle \rho_2, [x \texttt{ := } m\texttt{@}a(y'\texttt{@}b)]_w\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho_0[y \mapsto \rho_2(y')], v\rangle \cdot \langle \rho_2, w\rangle \cdot S \parallel \mu\rangle\\
&\langle\langle \rho_1, [\texttt{return } y\texttt{@}a]\rangle \cdot \langle \rho_2, [x \texttt{ := } m\texttt{@}a(y'\texttt{@}b)]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho_2[x \mapsto \rho_1(y)], v\rangle \cdot S \parallel \mu\rangle\ \dagger\\
&\langle\langle \rho, [x \texttt{ := null}]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto 0], v\rangle \cdot S \parallel \mu\rangle\\
&\langle\langle \rho, [x \texttt{ := new}(\overline{f})]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto \textsc{new}(\mu)], v\rangle \cdot S \parallel \mu[\textsc{new}(\mu) \mapsto \overline{[f_i \mapsto \texttt{null}]}]\rangle\\
&\langle\langle \rho, [x \texttt{ := } y \wedge z]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto \textsc{and}(\rho(y), \rho(z))], v\rangle \cdot S \parallel \mu\rangle\\
&\langle\langle \rho, [x \texttt{ := } y \vee z]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto \textsc{or}(\rho(y), \rho(z))], v\rangle \cdot S \parallel \mu\rangle\\
&\langle\langle \rho, [x \texttt{ := } y.f]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto \mu(\rho(y))(f)], v\rangle \cdot S \parallel \mu\rangle\\
&\langle\langle \rho, [x.f \texttt{ := } y]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho, v\rangle \cdot S \parallel \mu[\rho(x) \mapsto [f \mapsto \rho(y)]]\rangle\\
&\langle\langle \rho, [\texttt{main}]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho_0, v\rangle \cdot S \parallel \mu\rangle
\end{aligned}
$$

Figure 5 Small-step semantics rules that hold when $u \xrightarrow{p} v$. † This particular rule only applies if either $a = \texttt{?}$ or $\rho_1(y) \in \textsc{conc}(a)$ (see Section 4).

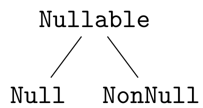

Figure 6 The ABST semilattice.

4 A Static Null-Pointer Analysis for PICL

In this section, we formalize a static null-pointer analysis, called NPA, for PICL on which we will build GNPA. Here, we will only consider completely annotated programs, $\textsc{Ann} = \{\texttt{Nullable}, \texttt{NonNull}\}$. Therefore, we use a "prime" symbol for sets like $\textsc{Inst}' \subseteq \textsc{Inst}$ to indicate that this is not the whole story. We present NPA's semilattice of abstract values, flow function, fixpoint algorithm, and how the analysis uses the results from the fixpoint algorithm to report warnings to the user.

4.1 Semilattice of Abstract Values

The set of abstract values $\textsc{Abst} = \{\texttt{Nullable}, \texttt{Null}, \texttt{NonNull}\}$ make up the finite semilattice defined in Figure 6. The partial order $\sqsubseteq\ \subseteq \textsc{Abst} \times \textsc{Abst}$ given is

$$\texttt{Null} \sqsubseteq \texttt{Nullable} \qquad \texttt{NonNull} \sqsubseteq \texttt{Nullable} \qquad \forall.\ l \in \textsc{Abst}\ .\ l \sqsubseteq l.$$

The join function $\sqcup : \textsc{Abst} \times \textsc{Abst} \to \textsc{Abst}$ induced by the partial order is:

$$\texttt{Null}\ \sqcup\ \texttt{NonNull} = \texttt{Nullable} \qquad \forall.\ l \in \textsc{Abst}\ .\ l\ \sqcup\ \texttt{Nullable} = \texttt{Nullable}$$

$$\forall.\ l \in \textsc{Abst}\ .\ l\ \sqcup\ l = l$$

Clearly, `Nullable` is the top element $\top$. Next, we relate this semilattice to $\textsc{Val}$ via a concretization function $\textsc{conc} : \textsc{Abst} \to \mathcal{P}^+(\textsc{Val})$:

$$\textsc{conc}(\texttt{Nullable}) = \textsc{Val}, \quad \textsc{conc}(\texttt{Null}) = \{0\}, \quad \textsc{conc}(\texttt{NonNull}) = \textsc{Val} \setminus \{0\},$$

which satisfies the property $\forall.\ l_1, l_2 \in \textsc{Abst}\ .\ l_1 \sqsubseteq\ l_2 \iff \textsc{conc}(l_1) \subseteq \textsc{conc}(l_2)$.

$$
\begin{aligned}
\textsc{flow}(x \texttt{ := } y, \sigma) &= \sigma[x \mapsto \sigma(y)] \\
\textsc{flow}(\texttt{branch } x, \sigma) &= \sigma \\
\textsc{flow}(\texttt{if } x, \sigma) &= \sigma[x \mapsto \texttt{NonNull}] \\
\textsc{flow}(\texttt{else } x, \sigma) &= \sigma[x \mapsto \texttt{Null}] \\
\textsc{flow}(x \texttt{ := } m\texttt{@}a(y\texttt{@}b), \sigma) &= \sigma[x \mapsto a] \\
\textsc{flow}(\texttt{proc } m\texttt{@}a(y\texttt{@}b), \sigma) &= \sigma_0[y \mapsto b] \\
\textsc{flow}(x \texttt{ := null}, \sigma) &= \sigma[x \mapsto \texttt{Null}] \\
\textsc{flow}(x \texttt{ := new}(\overline{f}), \sigma) &= \sigma[x \mapsto \texttt{NonNull}] \\
\textsc{flow}(x \texttt{ := } y \wedge z, \sigma) &= \begin{cases} \sigma[x \mapsto \texttt{Null}] & \text{if } \texttt{Null} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \texttt{Nullable}] & \text{if } \texttt{Nullable} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \texttt{NonNull}] & \text{otherwise} \end{cases} \\
\textsc{flow}(x \texttt{ := } y \vee z, \sigma) &= \begin{cases} \sigma[x \mapsto \texttt{NonNull}] & \text{if } \texttt{NonNull} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \texttt{Nullable}] & \text{if } \texttt{Nullable} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \texttt{Null}] & \text{otherwise} \end{cases} \\
\textsc{flow}(x \texttt{ := } y.f, \sigma) &= \sigma[x \mapsto \texttt{Nullable}][y \mapsto \texttt{NonNull}] \\
\textsc{flow}(x.f \texttt{ := } y, \sigma) &= \sigma[x \mapsto \texttt{NonNull}] \\
\textsc{flow}(\texttt{main}, \sigma) &= \sigma_0
\end{aligned}
$$

Figure 7 All consequential cases of the flow function used by NPA.

4.2 Flow Function

Similar to how we use $\textsc{Env}$ to represent mappings from variables to concrete values, we will use $\sigma \in \textsc{Map} = \textsc{Var} \rightharpoonup \textsc{Abst}$ to represent mappings from variables to abstract values—*abstract states*. Then, we extend the semilattice's partial order relation to abstract states $\sigma_1, \sigma_2 \in \textsc{Map}$:

$$\sigma_1 \sqsubseteq \sigma_2 \quad \iff \quad \forall.\ x \in \textsc{Var}\ .\ \sigma_1(x) \sqsubseteq \sigma_2(x)$$

We also extend the join operation to abstract states $\sigma_1, \sigma_2 \in \textsc{Map}$:

$$(\sigma_1 \sqcup \sigma_2)(x) = \begin{cases} a \sqcup b & \text{if } \sigma_1(x) = a \text{ and } \sigma_2(x) = b \\ a & \text{if } \sigma_1(x) = a \text{ and } \sigma_2(x) \text{ is undefined} \\ b & \text{if } \sigma_1(x) \text{ is undefined and } \sigma_2(x) = b \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The NPA's flow function $\textsc{flow} : \textsc{Inst}' \times \textsc{Map} \to \textsc{Map}$ is defined in Figure 7. Note, $\sigma_0 = \{x \mapsto \texttt{Null} : x \in \textsc{Var}\}$. Also, we omit the **return** $y\texttt{@}a$ case because it does not have CFG successors in a well-formed program.

4.2.1 Properties

It can be shown that this flow function is monotonic: for any $\iota \in \textsc{Inst}'$ and abstract states $\sigma_1, \sigma_2 \in \textsc{Map}$, if $\sigma_1 \sqsubseteq \sigma_2$ then $\textsc{flow}[\![\iota]\!](\sigma_1) \sqsubseteq \textsc{flow}[\![\iota]\!](\sigma_2)$. It can also be shown that the flow function is locally sound, *i.e.* the flow function models the concrete semantics at each step. To express this property formally, we define the predicate $\textsc{desc}(\rho, \sigma)$ on $\textsc{Env} \times \textsc{Map}$, which says that the abstract state σ "describes" the concrete environment ρ:

$$\textsc{desc}(\rho, \sigma) \quad \iff \quad \text{for all } x \in \textsc{Var} \;.\; \rho(x) \in \textsc{conc}(\sigma(x)).$$

Then, if $\langle S' \cdot \langle \rho, [\iota]_v \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho', v' \rangle \cdot S \parallel \mu' \rangle$, it must be the case that

$$\textsc{desc}(\rho, \sigma) \quad \implies \quad \textsc{desc}(\rho', \textsc{flow}[\![\iota]\!](\sigma)) \quad \text{for all} \quad \sigma \in \textsc{Map}.$$

4.3 Fixpoint Algorithm

This brings us to Algorithm 1 [15], which is used to analyze a program and compute whether each program variable is `Nullable`, `NonNull`, or `Null` at each program point (the program results π). More specifically, the algorithm applies the flow function to each program instruction recording or updating the results until a fixpoint is reached—*i.e.* until the results stop changing (becoming more approximate). The algorithm will always reach a fixpoint (terminate), because $\textsc{flow}$ is monotone and the height of the semilattice (Sec. 4.1) is finite. Note, the algorithm does not specify the order in which instructions are analyzed, because the order does not affect the results when $\textsc{flow}$ is monotonic. An implementation may choose to analyze instructions in CFG order—following the directed edges of the CFG.

Algorithm 1 Kildall's worklist algorithm

function $\textsc{Kildall}(\textsc{flow}, \sqcup, p)$
    $\pi \leftarrow \{v \mapsto \varnothing : v \in \textsc{Vert}_p\}$
    $V \leftarrow \textsc{Vert}_p$ $\quad\triangleright\ V \subseteq \textsc{Vert}_p$
    while $V \neq \varnothing$ **do**
        $[\iota]_v \leftarrow$ an element of V $\quad\triangleright\ v \in V$ and $\iota = \textsc{inst}_p(v)$
        $V \leftarrow V \setminus \{v\}$ $\quad\triangleright\ v \notin V$
        $\sigma \leftarrow \pi(v)$
        $\sigma' \leftarrow \textsc{flow}[\![\iota]\!](\sigma)$
        for $v \xrightarrow{p} u$ **do** $\quad\triangleright\ u \in \textsc{Vert}_p$
            if $\sigma' \sqcup \pi(u) \neq \pi(u)$ **then** $\quad\triangleright$ think of as $\sigma' \not\sqsubseteq \pi(u)$
                $\pi(u) \leftarrow \pi(u) \sqcup \sigma'$
                $V \leftarrow V \cup \{u\}$
            end if
        end for
    end while
    return π
end function

4.4 Safety Function & Static Warnings

Next, we present a way to use analysis results π produced by the fixpoint algorithm to determine whether to accept or reject a given program. Our goal is to ensure that when we run the program, it will not get stuck; that is, for any state ξ that the program reaches, we want to ensure that either ξ is a final state $\langle E \cdot \mathsf{nil} \parallel \mu \rangle$ or there is another state ξ' such that $\xi \longrightarrow_p \xi'$. To do this, we define the safety function $\textsc{safe}[\![\iota]\!](x) : \textsc{Inst}' \times \textsc{Var} \to \textsc{Abst}$, which returns the abstract value representing the set of "safe" values x can take on before

$$\begin{aligned}
\textsc{safe}(x \texttt{ := } m@a(y@b), y) &= b \\
\textsc{safe}(\texttt{return } y@a, y) &= a \\
\textsc{safe}(x \texttt{ := } y.f, y) &= \texttt{NonNull} \\
\textsc{safe}(x.f \texttt{ := } y, x) &= \texttt{NonNull}
\end{aligned}$$

Figure 8 All nontrivial cases of the safety function.

ι is executed. Figure 8 gives a few representative cases for SAFE, and in all the cases not shown SAFE returns `Nullable`. In particular, a procedure call's argument must adhere to the procedure's parameter annotation, a return value must adhere to its corresponding return annotation, and all field accesses must have non-null receivers. Therefore, the safety function guards against all undefined behavior.

4.4.1 Static Warnings

Now, we can state the meaning of a valid program $p \in \textsc{Prog}'$:

$$\text{for all} \quad [\iota]_v \in \textsc{Vert}_p \quad \text{and} \quad x \in \textsc{Var}\ . \quad \pi(v) = \sigma \quad \Longrightarrow \quad \sigma(x) \sqsubseteq \textsc{safe}[\![\iota]\!](x)$$

$$\text{where} \quad \pi = \textsc{Kildall}(\textsc{flow}, \sqcup, p).$$

That is, NPA emits static warnings when the fixpoint results disagree, according to the partial order $\sqsubseteq$, with the safety function. Also, we prove in Section 4.5 that a valid program does not get stuck.

4.5 Soundness of NPA

As discussed above, PICL's semantics are designed to get stuck when procedure annotations are violated or when null objects are dereferenced. Therefore, informally *soundness* says that a valid program does not get stuck during execution. Formally, soundness is defined with progress and preservation statements. Before their statement we must first define the notion of valid states to complement our definition of valid programs:

Let $p \in \textsc{Prog}'$. A state $\xi = \langle\langle \rho_1, v_1\rangle \cdot \langle \rho_2, v_2\rangle \cdots \langle \rho_n, v_n\rangle \cdot \mathsf{nil} \parallel \mu\rangle \in \textsc{State}_p$ is *valid* if

$$\text{for all} \quad 1 \le i \le n\ . \quad \textsc{desc}(\rho_i, \pi(v_i)) \quad \text{where } \pi = \textsc{Kildall}(\textsc{flow}, \sqcup, p).$$

A state is *valid* if it is described by the static analysis results π.

▶ **Proposition 1** (static progress). *Let $p \in \textsc{Prog}'$ be valid. If $\xi = \langle E_1 \cdot E_2 \cdot S \parallel \mu\rangle \in \textsc{State}_p$ is valid then $\xi \longrightarrow_p \xi'$ for some $\xi' \in \textsc{State}_p$.*

▶ **Proposition 2** (static preservation). *Let $p \in \textsc{Prog}'$ be valid. If $\xi \in \textsc{State}_p$ is valid and $\xi \longrightarrow_p \xi'$ then ξ' is valid.*

5 Gradual Null-Pointer Analysis

In this section, we derive GNPA from NPA, presented previously (Sec. 4). We proceed following the Abstracting Gradual Typing methodology introduced by Garcia *et al.* [13] in the context of gradual type systems, adapting it to fit the concepts of static analysis.

We present the GNPA's lifted semilattice (Sec. 5.1), flow and safety functions (Sec. 5.2), and fixpoint algorithm (Sec. 5.3). We also discuss how static (Sec. 5.4) and run-time warnings (Sec. 5.5) are generated by the analysis. Finally, Section 5.6 establishes the main properties of GNPA.

Note, here, annotations may be missing, so we extend our set of annotations with `?`: $\textsc{Ann} = \{\texttt{NonNull}, \texttt{Nullable}\} \cup \{\texttt{?}\}$.

5.1 Lifting the Semilattice

In this section, we lift the semilattice $(\textsc{Abst}, \sqsubseteq, \sqcup)$ (Sec. 4.1) by following the Abstracting Gradual Typing (AGT) framework [13]. First, we extend the set of semilattice elements $\textsc{Abst}$ to the new set $\widetilde{\textsc{Abst}} \supseteq \textsc{Abst}$:

$$\widetilde{\textsc{Abst}} = \textsc{Abst} \cup \{\texttt{?}\} \cup \{a\texttt{?} : a \in \textsc{Abst}\} =$$
$$\{\texttt{Nullable}, \texttt{NonNull}, \texttt{Null}, \texttt{?}, \texttt{NonNull?}, \texttt{Null?}\}.$$

Note that we equate the elements `Nullable?` and `Nullable` in $\widetilde{\textsc{Abst}}$. In Section 5.1.1, we give the semantics of the new lattice elements resulting in $\top = \texttt{Nullable?} = \texttt{Nullable}$. If $\textsc{Abst}$ had a bottom element $\bot$, then $\bot = \bot\texttt{?}$ similarly.

The join $\sqcup$ and partial order $\sqsubseteq$ are also lifted to their respective counterparts $\widetilde{\sqcup}$ (Sec. 5.1.2) and $\widetilde{\sqsubseteq}$ (Sec. 5.1.3). The resulting lifted semilattice $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ with lifted relation $\widetilde{\sqsubseteq}$ underpins the optimism in GNPA.

5.1.1 Giving Meaning to Missing Annotations

A straightforward way to handle `?` would be to make it the top element $\texttt{?} = \top$ or the bottom element $\texttt{?} = \bot$ of NPA's semilattice. However, neither choice is sufficient for our goal:

- If $\texttt{?} = \bot$, then $\texttt{?} \sqsubseteq a$ for all $a \in \textsc{Abst}$ and $\textsc{conc}(\bot) = \varnothing$. As a result, if the return annotation of a procedure was `?`, then we could use the return value in any context without the analysis giving a warning. But, anytime an initialized variable is checked against the `?` annotation, such as checking the non-null return value y against the `?` return annotation $\texttt{NonNull} \sqsubseteq \texttt{?}$, the check will fail as $a \not\sqsubseteq \texttt{?}$ for all $a \in \textsc{Abst}\ .\ a \neq \bot$.
- If we let $\texttt{?} = \top$ then we have $a \sqsubseteq \texttt{?}$ for all $a \in \textsc{Abst}$. Therefore, we can pass any argument to a parameter annotated as `?` without the static part of GNPA giving a warning. But, if the return annotation of that procedure is `?`, then the analysis will produce false positives in caller contexts wherever the return value is dereferenced. In other words, our analysis would operate exactly as `PolyNull` for the example in Fig. 1, which is not ideal.

Our goal is to construct an analysis system that does not produce false positive static warnings when a developer omits an annotation. To achieve this, we draw on work in gradual typing [13]. We define the injective concretization function $\gamma : \widetilde{\textsc{Abst}} \to \mathcal{P}^+(\textsc{Abst})$ where $\widetilde{\textsc{Abst}} \supseteq \textsc{Abst}$ is the lifted semilattice element set (Sec. 5.1):

$$\gamma(a) = \{a\} \quad \text{for} \quad a \in \textsc{Abst}, \quad \gamma(\texttt{?}) = \textsc{Abst}, \quad \text{and} \quad \gamma(a\texttt{?}) = \{b \in \textsc{Abst} : a \sqsubseteq b\}.$$

An element in $\textsc{Abst}$ is mapped to itself as it can only represent itself. In contrast, `?` may represent any element in $\textsc{Abst}$ at all times to support optimism in all possible contexts. Further, $a\texttt{?}$ means "a or anything more general than it," in contrast to a gradual formula $\phi \wedge \texttt{?}$ that means "ϕ or anything more specific than it" [2]. As a result, $a\texttt{?}$ does not play the intuitive role of "supplying missing information," as it would in gradual verification. Instead,

$a?$ is simply an artifact of our construction, which is why the only element of $\textsc{Ann} \setminus \textsc{Abst}$ is $?$.

Then, if $\gamma(\widetilde{a}) \subseteq \gamma(\widetilde{b})$ for some $\widetilde{a}, \widetilde{b} \in \widetilde{\textsc{Abst}}$, we write $\widetilde{a} \lesssim \widetilde{b}$ and say that $\widetilde{a}$ *is more precise than* $\widetilde{b}$. Further, $\iota_1 \lesssim \iota_2$ means that 1) the two instructions are equal except for their annotations, and 2) the annotations in ι_1 are more precise than the corresponding annotations in ι_2.

5.1.2 Lifted Join $\widetilde{\sqcup}$

We begin by introducing a semilattice definition [9], which states that a semilattice is an algebraic structure $(S, \sqcup)$ where for all $x, y, z \in S$ the following hold:

- $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$ (associativity)
- $x \sqcup y = y \sqcup x$ (commutativity)
- $x \sqcup x = x$ (idempotency)

Then, we write $x \sqsubseteq y$ when $x \sqcup y = y$ and it can be shown this $\sqsubseteq$ is a partial order. Recall that NPA uses $\sqcup$ in Algorithm 1 to compute a fixpoint that describes the behavior of a program p. The fixpoint can only be reached when $\sqcup$ is idempotent. Similarly, $\sqcup$ must be commutative and associative so that program instructions can be analyzed in any order. Thus, our extended join operation $\widetilde{\sqcup} : \widetilde{\textsc{Abst}} \times \widetilde{\textsc{Abst}} \to \widetilde{\textsc{Abst}}$ must be associative, commutative, and idempotent making $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ a join-semilattice.

To define such a function we turn to insights from gradual typing [13]. We define an abstraction function $\alpha : \mathcal{P}^{+}(\textsc{Abst}) \to \widetilde{\textsc{Abst}}$, which forms a Galois connection with γ:

$$\alpha(\widehat{a}) = \gamma^{-1}\left(\bigcap_{\substack{\widetilde{b} \in \widetilde{\textsc{Abst}} \\ \gamma(\widetilde{b}) \supseteq \widehat{a}}} \gamma(\widetilde{b}) \right)$$

where, for $a \in \textsc{Abst}$, γ^{-1} is:

$$\gamma^{-1}(\{a\}) = a \qquad \gamma^{-1}(\textsc{Abst}) = ? \qquad \gamma^{-1}(\{b \in \textsc{Abst} : a \sqsubseteq b\}) = a?.$$

Then we define the join of $\widetilde{a}, \widetilde{b} \in \widetilde{\textsc{Abst}}$ as follows:

$$\widetilde{a} \,\widetilde{\sqcup}\, \widetilde{b} = \alpha(\{a \sqcup b : \ a \in \gamma(\widetilde{a}) \text{ and } b \in \gamma(\widetilde{b})\})$$

For example,

$$\begin{aligned}
\texttt{NonNull} \,\widetilde{\sqcup}\, ? &= \alpha(\{a \sqcup b : a \in \{\texttt{NonNull}\} \text{ and } b \in \textsc{Abst}\}) && (1)\\
&= \alpha(\{\texttt{NonNull},\ \texttt{Nullable}\}) && (2)\\
&= \gamma^{-1}\left(\gamma(\texttt{NonNull?}) \cap \gamma(?)\right) && (3)\\
&= \gamma^{-1}\left(\{\texttt{NonNull},\ \texttt{Nullable}\} \cap \textsc{Abst}\right) && (4)\\
&= \gamma^{-1}\left(\{\texttt{NonNull},\ \texttt{Nullable}\}\right) && (5)\\
&= \texttt{NonNull?} && (6)
\end{aligned}$$

That is, the join of all the $\textsc{Abst}$ elements represented by `NonNull` and `?` results in the set $\{$`NonNull`, `Nullable`$\}$ (1, 2). Applying α to this set is equivalent to applying γ^{-1} to $\gamma($`NonNull?`$) \cap \gamma(?)$ (3); because, the only $\widetilde{\textsc{Abst}}$ elements that represent both `NonNull` and `Nullable` are `NonNull?` and `?`. The intersection of $\gamma($`NonNull?`$)$ and $\gamma(?)$ is $\{$`NonNull`,

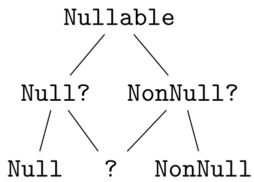

Figure 9 The semilattice structure induced by the lifted join $\widetilde{\sqcup}$. Specifically, this is the Hasse diagram of the partial order $\{(\widetilde{a}, \widetilde{b}) : \widetilde{a} \,\widetilde{\sqcup}\, \widetilde{b} = \widetilde{b}\}$.

`Nullable`} (4, 5), so we are really applying γ^{-1} to {`NonNull`, `Nullable`} (5). Therefore, `NonNull` $\widetilde{\sqcup}$ `?` $=$ `NonNull?` (6). Notice, the intersection of the representative sets γ(`NonNull?`) and γ(`?`) of {`NonNull`, `Nullable`} $= \widehat{a}$ is used to find the most precise element in $\widetilde{\text{ABST}}$ that can represent $\widehat{a}$.

Now we return to the properties of $\widetilde{\sqcup}$. Since $\sqcup$ is commutative, we have that $\widetilde{\sqcup}$ is commutative. Idempotency is also not too onerous: it is equivalent to the condition that every element of $\widetilde{\text{ABST}}$ represents a subsemilattice of ABST. That is, for every $\widetilde{a} \in \widetilde{\text{ABST}}$ and $a_1, a_2 \in \gamma(\widetilde{a})$, we must have $a_1 \sqcup a_2 \in \gamma(\widetilde{a})$. This is true by construction. Associativity is tricky and motivates our complex definition of $\widetilde{\text{ABST}}$. Ideally, $\widetilde{\text{ABST}}$ would be defined simply as ABST $\cup$ {`?`}, however in this case $\widetilde{\sqcup}$ is not associative:

$$
\begin{aligned}
\texttt{Null} \,\widetilde{\sqcup}\, (\texttt{NonNull} \,\widetilde{\sqcup}\, \texttt{?}) &= \texttt{Null} \,\widetilde{\sqcup}\, \texttt{?} \\
&= \texttt{?} \\
&\neq \texttt{Nullable} \\
&= \texttt{Nullable} \,\widetilde{\sqcup}\, \texttt{?} \\
&= (\texttt{Null} \,\widetilde{\sqcup}\, \texttt{NonNull}) \,\widetilde{\sqcup}\, \texttt{?}.
\end{aligned}
$$

Fortunately, our definition of $\widetilde{\text{ABST}}$ which also includes the intermediate optimistic elements `NonNull?` and `Null?` results in an associative $\widetilde{\sqcup}$ function and a finite-height semilattice $(\widetilde{\text{ABST}}, \widetilde{\sqcup})$. Figure 9 shows the semilattice structure induced by $\widetilde{\sqcup}$.

5.1.3 Lifted Order $\widetilde{\sqsubseteq}$

Now it is fairly straightforward to construct $\widetilde{\sqsubseteq}$. Recall, NPA emits static warnings when the fixpoint results disagree with the safety function, according to the partial order $\sqsubseteq$. The fixpoint results and the safety function now return elements in $\widetilde{\text{ABST}}$, so we lift $\sqsubseteq$ to $\widetilde{\sqsubseteq} \subseteq \widetilde{\text{ABST}} \times \widetilde{\text{ABST}}$ using the concretization function γ:

$$\widetilde{a} \,\widetilde{\sqsubseteq}\, \widetilde{b} \quad\Longleftrightarrow\quad \exists\,. \quad a \in \gamma(\widetilde{a}) \quad \text{and} \quad b \in \gamma(\widetilde{b}) \quad \text{such that} \quad a \sqsubseteq b \quad \text{for} \quad \widetilde{a}, \widetilde{b} \in \widetilde{\text{ABST}}.$$

Figure 10 gives the lifted order relation $\widetilde{\sqsubseteq}$ in graphical form.

The $\widetilde{\sqsubseteq}$ predicate is a maximally permissive version of the $\sqsubseteq$ predicate for `NonNull?`, `Null?`, and `?`. For example, `?` $\widetilde{\sqsubseteq}$ `NonNull` since γ(`?`) $=$ {`NonNull`, `Null`, `Nullable`}, γ(`NonNull`) $=$ {`NonNull`}, and `NonNull` $\sqsubseteq$ `NonNull`. By similar reasoning, `NonNull` $\widetilde{\sqsubseteq}$ `?`. In fact, `?` $\widetilde{\sqsubseteq}$ a $\widetilde{\sqsubseteq}$ `?`, `NonNull?` $\widetilde{\sqsubseteq}$ a $\widetilde{\sqsubseteq}$ `NonNull?`, and `Null?` $\widetilde{\sqsubseteq}$ a $\widetilde{\sqsubseteq}$ `Null?` for $a \in$ ABST. So, clearly $\widetilde{\sqsubseteq}$ is not a partial order. The $\widetilde{\sqsubseteq}$ predicate must be maximally permissive to support the optimism used in the `safeReverse` example from Figure 1 (Sec. 2.2): calls to `safeReverse` with null and non-null arguments are valid and dereferences of its return values are also valid. However, $\widetilde{\sqsubseteq}$ is the same as $\sqsubseteq$ when both of its arguments come from ABST,

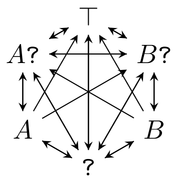

Figure 10 The lifted partial order, where each directed edge $\widetilde{a} \to \widetilde{b}$ means $\widetilde{a} \mathrel{\widetilde{\sqsubseteq}} \widetilde{b}$. (Self-loops are omitted). Here, `Nullable` is abbreviated $\top$, and `Null` and `NonNull` are abbreviated A and B respectively.

e.g. `NonNull` $\widetilde{\sqsubseteq}$ `Nullable` and `Nullable` $\widetilde{\not\sqsubseteq}$ `NonNull`. This allows our gradual analysis to apply NPA where annotations are complete enough to support it.

5.1.4 Properties

We previously mentioned some of the properties which $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ satisfy. Here, we formally state them, and their proofs can be found in the Appendix.

▶ **Proposition 3.** *$(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ is a semilattice; in other words, $\widetilde{\sqcup}$ is associative, idempotent, and commutative.*

▶ **Proposition 4.** *If the height of $(\textsc{Abst}, \sqcup)$ is $n > 0$, then the height of $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ is $n+1$* (i.e. *$(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ has finite-height*).

5.2 Lifting the Flow & Safety Functions

Now both instructions and abstract states ($\widetilde{\sigma} \in \widetilde{\textsc{Map}} = \textsc{Var} \rightharpoonup \widetilde{\textsc{Abst}}$) may contain optimistic abstract values. Therefore, similar to lifting the join $\widetilde{\sqcup}$, we follow the AGT *consistent function lifting* approach [13] when defining GNPA's flow function $\widetilde{\textsc{flow}} : \textsc{Inst} \times \widetilde{\textsc{Map}} \to \widetilde{\textsc{Map}}$ for this new domain.

Specifically, for $\iota \in \textsc{Inst}$ and $\widetilde{\sigma} = \{x \mapsto \widetilde{a}_x : x \in \textsc{Var}\} \in \widetilde{\textsc{Map}}$, we define

$$
\begin{aligned}
\widetilde{\textsc{flow}}[\![z \texttt{ := } m\texttt{@}a(y\texttt{@}b)]\!](\widetilde{\sigma}) &= \{x \mapsto \alpha(\{(\textsc{flow}[\![z \texttt{ := } m\texttt{@}a'(y\texttt{@}b')]\!](\sigma'))(x) \\
&\qquad : a' \in \gamma(a) \wedge b' \in \gamma(b) \wedge \sigma' \in \Sigma\}) : x \in \textsc{Var}\} \\
\widetilde{\textsc{flow}}[\![\texttt{proc } m\texttt{@}a(y\texttt{@}b)]\!](\widetilde{\sigma}) &= \{x \mapsto \alpha(\{(\textsc{flow}[\![\texttt{proc } m\texttt{@}a'(y\texttt{@}b')]\!](\sigma'))(x) \\
&\qquad : a' \in \gamma(a) \wedge b' \in \gamma(b) \wedge \sigma' \in \Sigma\}) : x \in \textsc{Var}\} \\
\widetilde{\textsc{flow}}[\![\iota]\!](\widetilde{\sigma}) &= \{x \mapsto \alpha(\{(\textsc{flow}[\![\iota]\!](\sigma'))(x) : \sigma' \in \Sigma\}) : x \in \textsc{Var}\} \quad \text{otherwise}
\end{aligned}
$$

$$
\text{where} \quad \Sigma = \{\{x \mapsto a_x : x \in \textsc{Var}\} : a_x \in \gamma(\widetilde{a}_x) \text{ for all } x \in \textsc{Var}\}.
$$

Note that the procedure call and procedure entry instructions are the only instructions in $\textsc{flow}$'s domain that may contain `?` annotations, so the corresponding $\textsc{flow}$ rules are lifted with respect to those annotations. Similarly, all rules are lifted with respect to their abstract states.

Recall that we defined the predicate $\textsc{desc}$ on $\textsc{Env} \times \textsc{Map}$ to express the local soundness of $\textsc{flow}$. For $\widetilde{\textsc{flow}}$, we lift $\textsc{desc}$ to $\widetilde{\textsc{desc}}$ on $\textsc{Env} \times \widetilde{\textsc{Map}}$ such that it is maximally permissive like the $\widetilde{\sqsubseteq}$ predicate:

$$
\widetilde{\textsc{desc}}(\rho, \widetilde{\sigma}) \quad \iff \quad \textsc{desc}(\rho, \sigma) \text{ for some } \sigma \in \Sigma
$$

where Σ is constructed in the same way as for $\widetilde{\text{FLOW}}$.

Finally, we again follow the consistent function lifting methodology to construct $\widetilde{\text{SAFE}}$: $\text{INST} \times \text{VAR} \to \widetilde{\text{ABST}}$ from $\text{SAFE} : \text{INST}' \times \text{VAR} \to \text{ABST}$:

$$\widetilde{\text{SAFE}}[\![z \texttt{ := } m@a(y@b)]\!](x) = \alpha(\{\text{SAFE}[\![z \texttt{ := } m@a'(y@b')]\!](x) : a' \in \gamma(a) \wedge b' \in \gamma(b)\})$$
$$\widetilde{\text{SAFE}}[\![\texttt{proc } m@a(y@b)]\!](x) = \alpha(\{\text{SAFE}[\![\texttt{proc } m@a'(y@b')]\!](x) : a' \in \gamma(a) \wedge b' \in \gamma(b)\})$$
$$\widetilde{\text{SAFE}}[\![\texttt{return } y@a]\!](x) = \alpha(\{\text{SAFE}[\![\texttt{return } y@a']\!](x) : a' \in \gamma(a)\})$$
$$\widetilde{\text{SAFE}}[\![\iota]\!](x) = \alpha(\text{SAFE}[\![\iota]\!](x)) \quad \text{otherwise}$$

Other than the casewise-defined FLOW rules for $\wedge$ and $\vee$, the lifted $\widetilde{\text{FLOW}}$ and $\widetilde{\text{SAFE}}$ functions simplify down to the same computation rules as FLOW and SAFE as shown in Figure 7 and Figure 8 respectively, replacing FLOW with $\widetilde{\text{FLOW}}$ and SAFE with $\widetilde{\text{SAFE}}$.

5.3 Lifting the Fixpoint Algorithm

To lift the fixpoint algorithm, we simply plug $\widetilde{\text{FLOW}}$ and $\widetilde{\sqcup}$ into Algorithm 1 to compute $\widetilde{\pi} = \text{KILDALL}(\widetilde{\text{FLOW}}, \widetilde{\sqcup}, p) : \text{VERT}_p \to \widetilde{\text{MAP}}$ for any $p \in \text{PROG}$.

5.4 Static Warnings

Using the lifted safety function, we say that a partially-annotated program $p \in \text{PROG}$ is *statically valid* if

$$\text{for all} \quad [\iota]_v \in \text{VERT}_p \quad \text{and} \quad x \in \text{VAR}, \quad \widetilde{\pi}(v) = \widetilde{\sigma} \quad \Longrightarrow \quad \widetilde{\sigma}(x) \mathrel{\widetilde{\sqsubseteq}} \widetilde{\text{SAFE}}[\![\iota]\!](x)$$

where $\widetilde{\pi} = \text{KILDALL}(\widetilde{\text{FLOW}}, \widetilde{\sqcup}, p)$.

Each piece of GNPA's static system $((\widetilde{\text{ABST}}, \widetilde{\sqcup}), \widetilde{\sqsubseteq}, \widetilde{\text{FLOW}}, \widetilde{\text{SAFE}}$, and the fixpoint algorithm) is designed to be maximally optimistic for missing annotations. Therefore, the resulting system will not produce false positive warnings due to missing annotations. The system is also designed to apply NPA where annotations are available to support it, so it will still warn about violations of procedure annotations or null object dereferences where possible. See Section 2.2 for more information.

5.5 Dynamic Checking

GNPA's static system reduces false positive warnings at the cost of soundness. For example, as in Section 2.3, the analysis may assume a variable with a `?` annotation is non-null to satisfy an object dereference when the variable is actually null. In order to avoid false negatives and ensure that our gradual analysis is sound, we modify the semantics of PICL to insert run-time checks where the analysis may be unsound. That is, if p is *statically valid* and there are program points $[\iota]_v$ such that

$$a \not\sqsubseteq \bigsqcup \gamma(\widetilde{\text{SAFE}}[\![\iota]\!](x)) \quad \text{for some} \quad x \in \text{VAR} \quad \text{and} \quad a \in \gamma((\widetilde{\pi}(v))(x)),$$

then a run-time check must be inserted at those points to ensure the value of x is in $\text{CONC}(\bigsqcup \gamma(\widetilde{\text{SAFE}}[\![\iota]\!](x)))$.

More precisely, we define a dedicated error state `error` and expand the set of run-time states to be $\widetilde{\text{STATE}}_p = \text{STATE}_p \cup \{\texttt{error}\}$. Then we define a restricted semantics $\widetilde{\longrightarrow}_p$ on $\widetilde{\text{STATE}}_p \times \widetilde{\text{STATE}}_p$ as follows. Let $\xi \in \text{STATE}_p$. If

$$\xi = \langle\langle \rho, [\iota] \rangle \cdot S \parallel \mu \rangle \quad \text{and} \quad \neg\widetilde{\text{DESC}}(\rho, \{x \mapsto \widetilde{\text{SAFE}}[\![\iota]\!](x) : x \in \text{VAR}\})$$

then $\xi \mathrel{\widetilde{\longrightarrow}_p}$ `error`. If there is some $\xi' \in \textsc{State}_p$ such that $\xi \longrightarrow_p \xi'$, then $\xi \mathrel{\widetilde{\longrightarrow}_p} \xi'$. Otherwise, there is no $\widetilde{\xi}' \in \widetilde{\textsc{State}}_p$ such that $\xi \mathrel{\widetilde{\longrightarrow}_p} \xi'$.

5.6 Gradual Properties

GNPA is sound, *conservative extension* of NPA—the static system is applied in full to programs with complete annotations, and adheres to the gradual guarantees inspired by Siek *et al.* [22]. The gradual guarantees ensure losing precision is harmless, *i.e.* increasing the number of missing annotations in a program does not break its validity or reducibility.

To formally present each property, we first extend the notion of a valid state. Let $p \in \textsc{Prog}$. A state $\xi = \langle\langle\rho_1, v_1\rangle \cdot \langle\rho_2, v_2\rangle \cdots \langle\rho_n, v_n\rangle \cdot \mathsf{nil} \parallel \mu\rangle \in \textsc{State}_p$ is valid if

$$\text{for all} \quad 1 \leq i \leq n, \quad \widetilde{\text{DESC}}(\rho_i, \widetilde{\pi}(v_i)) \quad \text{where} \quad \widetilde{\pi} = \textsc{Kildall}(\widetilde{\text{FLOW}}, \widetilde{\sqcup}, p).$$

Then, for fully-annotated programs, GNPA and the modified semantics are conservative extensions of NPA and PICL's semantics, respectively.

▶ **Proposition 5** (conservative static extension).
If $p \in \textsc{Prog}'$ then $\textsc{Kildall}(\text{FLOW}, \sqcup, p) = \textsc{Kildall}(\widetilde{\text{FLOW}}, \widetilde{\sqcup}, p)$.

▶ **Proposition 6** (conservative dynamic extension). *Let $p \in \textsc{Prog}'$ be valid, and let $\xi_1, \xi_2 \in \textsc{State}_p$. If ξ_1 is valid then $\xi_1 \longrightarrow_p \xi_2$ if and only if $\xi_1 \mathrel{\widetilde{\longrightarrow}_p} \xi_2$.*

GNPA is sound, *i.e.* valid programs will not get stuck during execution. However, programs may step to a dedicated `error` state when run-time checks fail. Soundness is stated with a progress and preservation argument.

▶ **Proposition 7** (gradual progress). *Let $p \in \textsc{Prog}$ be valid. If $\xi = \langle E_1 \cdot E_2 \cdot S \parallel \mu\rangle \in \textsc{State}_p$ is valid then $\xi \mathrel{\widetilde{\longrightarrow}_p} \widetilde{\xi}'$ for some $\widetilde{\xi}' \in \widetilde{\textsc{State}}_p$.*

▶ **Proposition 8** (gradual preservation). *Let $p \in \textsc{Prog}$ be valid. If $\xi \in \textsc{State}_p$ is valid and $\xi \mathrel{\widetilde{\longrightarrow}_p} \xi'$ for some $\xi' \in \textsc{State}_p$, then ξ' is valid.*

Finally, GNPA satisfies both the static and dynamic gradual guarantees. Both of the guarantees rely on a definition of *program precision*. Specifically, if programs p_1 and p_2 are identical except perhaps that some annotations in p_2 are `?` where they are not `?` in p_1, then we say that p_1 *is more precise than* p_2, and write $p_1 \lesssim p_2$.

Then, the *static gradual guarantee* states that increasing the number of missing annotations in a valid program does not introduce static warnings (*i.e.* break program validity).

▶ **Proposition 9** (static gradual guarantee). *Let $p_1, p_2 \in \textsc{Prog}$ such that $p_1 \lesssim p_2$. If p_1 is statically valid then p_2 is statically valid.*

The *dynamic gradual guarantee* ensures that increasing the number of missing annotations in a program does not change the observable behavior of the program (*i.e.* break program reducibility for valid programs).

▶ **Proposition 10** (dynamic gradual guarantee). *Let $p_1, p_2 \in \textsc{Prog}$ be statically valid, where $p_1 \lesssim p_2$. Let $\xi_1, \xi_2 \in \textsc{State}_{p_2}$. If $\xi_1 \mathrel{\widetilde{\longrightarrow}_{p_1}} \xi_2$ then $\xi_1 \mathrel{\widetilde{\longrightarrow}_{p_2}} \xi_2$.*

Note, the small-step semantics $\widetilde{\longrightarrow}$ are designed to make the proofs of the aforementioned properties easier at the cost of easily implementable run-time checks. Therefore, we give the following proposition that connects a more implementable design to $\widetilde{\longrightarrow}$. That is, we

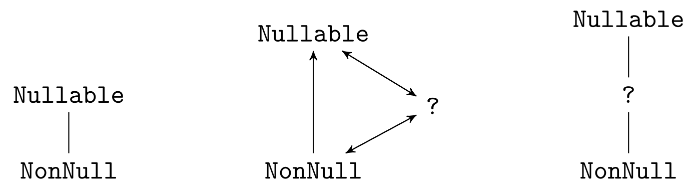

Figure 11 *Left:* The starting null-pointer semilattice for Graduator. *Middle:* The lifted partial ordering, where each directed edge $\widetilde{a} \to \widetilde{b}$ means $\widetilde{a} \mathrel{\widetilde{\sqsubseteq}} \widetilde{b}$. (Self-loops are omitted.) *Right:* The semilattice structure induced by the lifted join $\widetilde{\sqcup}$.

can use the contrapositive of this proposition to implement more optimal run-time checks. Specifically, the naïve implementation would check each variable at each program point to make sure it satisfies the safety function for the instruction about to be executed. But Proposition 1 tells us that we only need to check variables at runtime when our analysis results don't already guarantee (statically) that they will satisfy the safety function.

▶ **Proposition 11** (run-time checks). *Let $p \in \textsc{Prog}$ be valid according to $\widetilde{\pi} = \textsc{Kildall}(\widetilde{\mathit{FLOW}}, \widetilde{\sqcup}, p)$, and let $\xi = \langle\langle \rho, [\iota]_v\rangle \cdot S \parallel \mu\rangle \in \mathit{STATE}_p$ be valid. If $\xi \mathrel{\widetilde{\longrightarrow}_p} \texttt{error}$ then there is some $x \in \textsc{Var}$ and $a \in \gamma((\widetilde{\pi}(v))(x))$ such that $a \not\sqsubseteq \bigsqcup \gamma(\widetilde{\mathit{SAFE}}[\![\iota]\!](x))$.*

6 Empirical Evaluation

In this section, we discuss the implementation of GNPA and two studies designed to evaluate its usefulness in practice. Preliminary evidence suggests that our analysis can be used at scale, produces less false positives than state-of-the-art tools, and eliminates on average more than half of the null-pointer checks Java automatically inserts at run time.

6.1 Research Questions

We seek answers to the following questions:

1. Can a gradual null-pointer analysis be effectively implemented and used at scale?
2. Does such a null-pointer analysis produce a reasonable number of false positives?
3. Does the gradual null-pointer analysis perform significantly less null-pointer checks than the naïve approach of checking every dereference?

6.2 Prototype

Facebook Infer provides a framework to construct static analyses that use abstract interpretation. We built a prototype of GNPA, called *Graduator*[4], in this framework. Our prototype uses Infer's HIL intermediate language representation (IR). As a result, Graduator can be used to analyze code written in C, C++, Objective-C, and Java.

The preceding case study (Secs. 3–5) uses a base semilattice with three elements, `Null`, `NonNull`, and `Nullable`, in order to demonstrate that a semilattice lifting may contain additional intermediate optimistic elements, `Null?` and `NonNull?`. For simplicity, we implemented the semilattice from Figure 11, along with its lifted variant, order relation and join function, in our prototype. This semilattice is the same as the base one in the case study except it does not contain `Null`: the initial static semilattice has only `NonNull` and

[4] we will submit this as an artifact

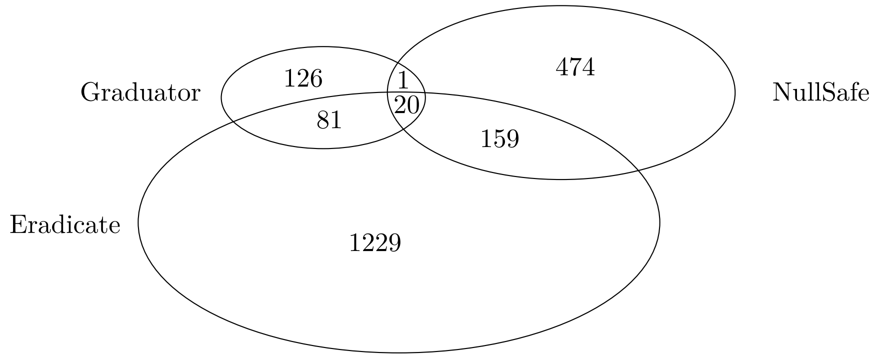

Figure 12 The total number of static warnings reported by the three Infer null checkers, for all 15 repositories.

`Null`, and the gradual semilattice only adds one additional `?` element. There are a couple other differences between our formalism and our Graduator prototype, one of which is that Graduator allows field annotations while our formalism does not.

Infer does not support modifying Java source code, so Graduator simply reports the locations where it should insert run-time checks rather than inserting them directly. In fact, Graduator may output any of the following:

- `GRADUAL_STATIC`—a static warning.
- `GRADUAL_CHECK`—a location to check a possibly-null dereference.
- `GRADUAL_BOUNDARY`—another location to insert a check, such as passing an argument to a method, returning from a method, or assigning a value to a field.

Since Java checks for null-pointer dereferences automatically, soundness is preserved. A more complete implementation of GNPA would insert run-time checks as part of the build process. As a result, some bugs may be caught earlier when the gradual analysis inserts checks at method boundaries and field assignments.

By implementing Graduator with Infer's framework, Graduator is guaranteed to operate at scale. We also evaluate Graduator on a number of open source repositories as discussed in Sections 6.3 and 6.4. Thus, the answer to RQ1 is yes.

6.3 Static Warnings

To evaluate Graduator, we ran it on 15 of the 18 open-source Java repositories used to evaluate NullAway [3]. We also successfully ran Infer's existing null-pointer checkers Eradicate and NullSafe on the repositories. Figure 12 shows the number of *static* warnings produced by each of these three checkers: 1489 for Eradicate, 654 for NullSafe, and 228 for Graduator, for a total of 2371.

Based on the NullAway paper (in which Uber states that in practice they have found no instances of null-pointer dereferences caused by their tool's unsoundness), it seems reasonable to assume that these repositories do not have null-pointer bugs, since NullAway itself reports no static warnings for these repositories. After examining all 2371 warnings ourselves, we found that all but 57 were false positives due to systematic imprecision in the analysis tools. We were unable to determine whether the remaining 57 warnings represent actual bugs or not.

Under this assumption, Graduator reports significantly fewer false positives than Infer's existing null-pointer checkers. Therefore, a gradual null-pointer analysis reports a reasonable number of false positives in practice (RQ2). An interesting aspect of Figure 12 is how many warnings are produced by only one of the checkers: 1229 for Eradicate, 474 for NullSafe, and 126 for Graduator. Many of these warnings arose from generated and test case code.

6.3.1 Generated Code

Several of the 15 repositories generate code as part of their build process, and in some cases, the analysis tools gave warnings about the generated code. This accounts for

- 380 of the warnings given by NullSafe alone,
- 356 of the warnings given by Eradicate alone,
- 130 of the warnings given by both Eradicate and NullSafe but not Graduator, and
- 8 of the warnings given by Graduator alone.

Graduator reports significantly fewer static warnings for generated code, because such code is typically unannotated and Graduator is designed to be optimistic when annotations are missing.

6.3.2 Test Code

It is reasonable to assume that test code does not contain null dereference bugs, because if it did, then those bugs would show up when the tests are run. Static warnings about test code account for

- 384 of the warnings given by Eradicate alone, and
- 73 of the warnings given by both Eradicate and Graduator, but not NullSafe.

That is, Graduator reports fewer warnings for test code than Eradicate, but more than NullSafe. The NullSafe checker does not appear to treat test code specially, so it is unclear why NullSafe is performing better than Graduator for such code.

6.3.3 Remaining False Positives

The reader may wonder why Graduator reports any false positives on this codebase, since it intuitively seems that the static portion of a gradual analysis ought to be optimistic. Examining the warnings given by Graduator, we see that none of the warnings are due to treating missing annotations pessimistically; instead, they are due to places where the analysis has whatever annotations it needs, but the analysis is imprecise in other respects. For example, one common source of false positives is when a field is checked for null, then is read again. Our original static analysis is limited in that it does not treat fields flow-sensitively, causing false positives that are independent of the choice to be gradual or not with respect to annotations.

NullAway avoids giving false positives on this same codebase, due to a combination of some unsound assumptions and a more precise analysis approach. While our approach for deriving gradual program analysis focuses on retaining soundness through a combination of static and dynamic checks, incorporating more precise analysis techniques (e.g. a flow-sensitive treatment of fields, perhaps in combination with a gradual alias analysis) could eliminate more of these false positives. In the meantime, our comparison to Eradicate and NullSafe is appropriate as these are the static analysis tools taking the most similar approach.

6.4 Run-time Checks

For the same set of 15 repositories analyzed by NullAway, we performed another experiment using our prototype. We configured Graduator to ignore *all* annotations, so in effect, every field, argument, and return value was annotated as `?`. For each repository, we counted all the

Table 1 Percentage of null-dereference checks which Graduator found to be redundant.

repository	dereference sites	eliminated checks	percent eliminated
keyvaluestore	419	156	37%
uLeak	620	241	39%
butterknife	2773	1129	41%
jib	5896	2499	42%
skaffold-tools-for-java	366	185	51%
picasso	2719	1458	54%
meal-planner	858	475	55%
caffeine	9455	5701	60%
AutoDispose	3218	1993	62%
ColdSnap	6360	4325	68%
ReactiveNetwork	2097	1626	78%
okbuck	19089	15130	79%
FloatingActionButtonSpeedDial	3049	2581	85%
QRContact	1272	1171	92%
OANDAFX	2216	2056	93%
overall	60407	40726	67%

locations where Graduator gave a `GRADUAL_STATIC`, `GRADUAL_CHECK`, or `GRADUAL_BOUNDARY` warning, and compared that number to the total number of pointer dereferences in the code. By ignoring annotations, we ensured that each of these warnings appeared on dereferences, rather than allowing early checks at, e.g., method boundaries. We also ran analogous experiments with annotations enabled, but the number of run-time check warnings found were very similar to the numbers found with annotations disabled.

Table 1 shows what percentage of these dereference sites received no static warnings or run-time checks. Recall that Java automatically checks all dereferences to ensure that they are not null. Because GNPA is sound, this figure shows the percentage of null checks that are provably redundant, and could be safely removed by an ahead-of-time compiler.

Since we were able to eliminate an average of 67% of the null checks which Java automatically inserts, this experiment suggests the answer to RQ3 is yes. Note that these numbers only discuss the number of dereferences that appear in the code, and do not take into account which of these dereferences are executed more or less frequently at run-time.

This also illustrates an important practical difference between GNPA and other null-pointer analyses. While a sound static analysis can be used to prove the redundancy of run-time checks, and an unsound static analysis can be used to reduce the number of false positives, neither of those can do both at the same time. On the other hand, a gradual analysis can both prove the redundancy of run-time checks and reduce reported false positives.

7 Related Work

As discussed previously, our work builds on prior research in gradual typing: the criteria for gradual type systems [22] and the Abstracting Gradual Typing methodology, which develops a gradual type system from a purely static one [13]. In contrast to prior work in gradual typing, we address the challenges of tracking transitive dataflow relationships, rather than the local checks of typical type systems. In doing so, we gradualize, for the first time, the abstract interpretation of a program [8], and the canonical dataflow analysis fix-point algorithm [15].

The most closely related work in program analysis consists of *hybrid analyses*, which combine static and dynamic analysis techniques to counteract the weaknesses inherent to each approach. For example, Choi *et al.* [7] used a static analysis to substantially lower the run-time overhead of a dynamic data race analysis. Prior work on hybrid program analyses combines static and dynamic techniques in ad-hoc ways. Instead, we propose a principled methodology for deriving a hybrid (gradual) analysis from a static one, and show that the resulting analysis adheres to desirable properties such as soundness and the gradual guarantee.

There is a large body of literature on static program analysis, including multiple specialized conferences. Our work opens the door to gradual versions of them. Previously, we discussed existing null-pointer analysis tools [10], [3] and frameworks [20], and how GNPA is an improvement over them. Notably, our prototype is implemented in Infer's framework [10].

The Granullar type system [5] and the Blame for Null calculus[18] are gradual type systems for nullness, and thus solve a related problem to GNPA. The main difference in our work is that we use dataflow analysis instead of typing. This results in a significantly different user experience, as a full static specification within a gradual type system typically requires many more types to be specified (e.g. on all local variables) compared to a dataflow analysis, where for example we do not require (or even allow) nullity annotations on local variables. Basing our work on dataflow analysis also has a major impact on the technical development, requiring the novel lattice-based gradualization framework described in this paper rather than the well-known type-based gradualization approaches used in Granullar and Blame for Null. Blame for Null also investigates the notion of blame, which we leave for future work in the program analysis setting.

Contract checking [17, 12] can be used to check properties like nullness. Building on the idea of hybrid type checking [16], Xu *et al.* [24] explored how to check contracts using a hybrid of static and dynamic analysis. Their work was specialized to the context of logical assertions, whereas we are in the area of lattice-based program analyses. It is also unclear whether their approach conforms to the gradual guarantee.

O'Hearn *et al.* [19] proposed Incorrectness Logic as a means of proving that a program has a bug, rather than proving it correct. This is consistent with our goal of reducing false positives, but it stays in the realm of static reasoning, and therefore gives up soundness. In contrast, we reduce false positives without giving up soundness by adding run-time checks.

8 Conclusion

This paper is the first work on gradual program analysis. We introduced a framework which transforms abstract interpretation based static analyses relying on annotations into gradual ones. Gradual analyses handle missing annotations specially, allowing them to smoothly leverage both static and dynamic techniques. Static information is used where possible and dynamic information where necessary to reduce false positives while preserving soundness. Such analyses are also *conservative extensions* of their underlying static analyses and adhere to *gradual guarantees*, which state that losing precision is harmless. When presenting our framework, we developed a gradual null-pointer analysis, GNPA, with the previously mentioned properties that reduces false positives compared to existing tools.

Importantly, the gradual framework can be applied as described to any abstract interpretation based static analysis under the following restrictions. The analysis should support annotations, have a finite-height semilattice, a monotonic, locally-sound flow function, a safety function, and operate on a first-order, procedural, imperative programming language. Additionally, checking membership in the semilattice should be decidable. Finally, we do not support widening, but we do support context-sensitivity. In the future, we plan to explore extensions of our framework for infinite-height semilattices and widening.

On the empirical side, there are further research questions to be answered: How often does a gradual analysis catch bugs statically versus how often does it catch them at run time? Is performance lost or gained when run time checks are inserted earlier via annotations rather than just-in-time? Finally, a gradual analysis will still report false positives anywhere its base static analysis is utilized and reports false positives. As a result, we plan to explore the aforementioned research questions, including the trade-off between gradual analyses reducing false positives and being conservative extensions of underlying static analyses.

References

1 Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252, 2010.

2 Johannes Bader, Jonathan Aldrich, and Éric Tanter. Gradual program verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 25–46. Springer, 2018.

3 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical type-based null safety for java. *arXiv preprint arXiv:1907.02127*, 2019.

4 Mike Barnett, Manuel Fahndrich, Francesco Logozzo, and Diego Garbervetsky. Annotations for (more) precise points-to analysis. 2007.

5 Dan Brotherston, Werner Dietl, and Ondřej Lhoták. Granullar: Gradual nullable types for java. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 87–97, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3033019.3033032`, `doi:10.1145/3033019.3033032`.

6 Patrice Chalin and Perry R James. Non-null references by default in java: Alleviating the nullity annotation burden. In *European Conference on Object-Oriented Programming*, pages 227–247. Springer, 2007.

7 Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert OCallahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI 02, page 258269, New York, NY, USA, 2002. Association for Computing Machinery. `doi:10.1145/512529.512560`.

8 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 77)*, pages 238–252, Los Angeles, CA, USA, January 1977.

9 Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

10 Facebook. Infer: A tool to detect bugs in java and c/c++/objective-c code before it ships. `https://fbinfer.com/`, 2019. Accessed: 2019-10-28.

11 Facebook. Eradicate. `https://fbinfer.com/docs/checker-eradicate`, 2020. Accessed: 2021-1-10.

12 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2002)*, pages 48–59, Pittsburgh, PA, USA, September 2002.

13 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. ACM. URL: `http://doi.acm.org/10.1145/2837614.2837670`, `doi:10.1145/2837614.2837670`.

14 Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

15 Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

16 Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):1–34, 2010.

17 Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

18 Abel Nieto, Marianna Rapoport, Gregor Richards, and Ondřej Lhoták. Blame for null. In *European Conference on Object-Oriented Programming*, 2020.

19 Peter W O'Hearn. Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.

20 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, 2008.

21 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

22 Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

23 Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

24 Dana N Xu. Hybrid contract checking via symbolic simplification. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pages 107–116, 2012.

A Appendix

A.1 Proofs

These proofs apply generally to any particular language/semilattice/analysis that fits within the bounds of our formal framework, of which the GNPA formalism detailed in the paper is just a particular example. We left out a few formal details in the main body of the paper, for presentation's sake; we now formalize those missing details, before proceeding to the proofs.

- Our case study language declares programs $p \in \textsc{Prog}$ to satisfy the following well-formedness rules:
  1. *Unique entry point to the program:* There exists exactly one node $v_0 \in \textsc{Vert}_p$ such that $\textsc{inst}_p(v_0) = (\texttt{main})$. This node has no predecessors and serves as the entry point to p.
  2. *Every node belongs to exactly one procedure, or to* **`main`***:* Let $\textsc{descend} : \textsc{Vert}_p \to \mathcal{P}^+(\textsc{Vert}_p)$ give the descendants of each node in the control flow graph. The set $\{\textsc{descend}(v_0)\} \cup \{\textsc{descend}(\textsc{proc}(m)) : m \in \textsc{Proc}\}$ is a partition of $\textsc{Vert}_p$.
  3. *Always a path to return from a procedure:* For each $u \in \textsc{Vert}_p$ there exists at least one node $[\texttt{return}\ y\texttt{@}a]_v \in \textsc{descend}(u)$. If $v \in \textsc{descend}(\texttt{proc}\ m\texttt{@}a'(y\texttt{@}b))$ then each such v must have $a = a'$.
  4. *Call sites agree with procedure annotations:* For each $[x \texttt{ := } m\texttt{@}a(y\texttt{@}b)]$, the annotations must match the procedure signature $\textsc{proc}(m) = \texttt{proc}\ m\texttt{@}a(y'\texttt{@}b)$.
  5. For every $[\iota]_u \in \textsc{Vert}_p$:

a. *Always a branch to follow:* If $\iota = \texttt{branch}\ y$ then u has exactly two successors $[\texttt{if}\ y]$ and $[\texttt{else}\ y]$.

b. *No dead code after return:* If $\iota = \texttt{return}\ y\texttt{@}a$ then u has no successors.

c. *Control flow is unique:* Otherwise u has exactly one successor that is not an if or else node.

- The property our safety function must satisfy is that given a state $\xi = \langle\langle\rho, [\iota]_v\rangle \cdot E \cdot S \parallel \mu\rangle$, if

$$\textsc{desc}(\rho, \{x \mapsto \textsc{safe}[\![\iota]\!](x) : x \in \textsc{Var}\})$$

then $\xi \longrightarrow_p \xi'$ for some $\xi' \in \textsc{State}_p$. Also, these safe values must come directly from the annotations.

- For any $\widehat{a} \in \mathcal{P}^+(\textsc{Abst})$ and $\widetilde{b} \in \widetilde{\textsc{Abst}}$,
  1. $\widehat{a} \subseteq \gamma(\alpha(\widehat{a}))$ ("soundness"), and
  2. $\widehat{a} \subseteq \gamma(\widetilde{b})$ implies $\alpha(\widehat{a}) \lesssim \widetilde{b}$ ("optimality").

- The associativity example in Section 5.1.2 shows that in some cases we need to make $\widetilde{\textsc{Abst}}$ a *strict* superset of $\{\texttt{Nullable}, \texttt{Null}, \texttt{NonNull}, \texttt{?}\}$, in order for $\widetilde{\sqcup}$ to be associative. One approach could be to define $\widetilde{\textsc{Abst}}$ to have an element for *every* subsemilattice of $\textsc{Abst}$; we will call this the "full lifting" of $\textsc{Abst}$. It can be shown that α always exists for the full lifting, and that $\widetilde{\sqcup}$ is always associative in the full lifting. Unfortunately, even if the height of $\textsc{Abst}$ is finite, the height of the full lifting is not necessarily finite; that is, if $\widetilde{\textsc{Abst}}$ is the full lifting then there can exist sequences $\widetilde{a}_1, \widetilde{a}_2, \ldots \in \widetilde{\textsc{Abst}}$ such that $\widetilde{a}_k \,\widetilde{\sqcup}\, \widetilde{a}_{k+1} = \widetilde{a}_{k+1}$ for all k.
  To address this, we will treat the full lifting as a sort of "universe," consider $\{\texttt{Nullable}, \texttt{Null}, \texttt{NonNull}, \texttt{?}\}$ to be a generating set, and let $\widetilde{\textsc{Abst}}$ be the subset of the full lifting generated by $\{\texttt{Nullable}, \texttt{Null}, \texttt{NonNull}, \texttt{?}\}$ under the operation $\widetilde{\sqcup}$. We show in subsection 5.1.4 that this is equivalent to saying

$$\widetilde{\textsc{Abst}} = \textsc{Abst} \cup \{?\} \cup \{a? : a \in \textsc{Abst}\} \quad \text{where} \quad \gamma(a?) = \{b \in \textsc{Abst} : a \sqsubseteq b\}.$$

  We will call this the "small lifting" of $\textsc{Abst}$, and it is the lifting we will use to construct gradual analyses. The abstraction function α always exists on the small lifting $\widetilde{\textsc{Abst}}$, and $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ is a finite-height semilattice; see subsection 5.1.4.

- We insist that it is always possible to annotate a program in a way that does not restrict its semantics. That is, for any program $p \in \textsc{Prog}$, there must exist a program $p' \in \textsc{Prog}'$ such that p' is the same as p except for replacing every instance of ? with $\top$ (a stronger condition than $p' \lesssim p$), and such that $\textsc{State}_{p'} = \textsc{State}_p$ and the semantics of p' are equal to the semantics of p.

Proposition 1:

Proof. Let $\pi = \textsc{Kildall}(\textsc{flow}, \sqcup, p)$. Then let $\langle\rho, [\iota]_v\rangle = E_1$ and $\sigma = \pi(v)$. Let $x \in \textsc{Var}$ such that $\rho(x) = d \in \textsc{Val}$. Because ξ is valid, $\rho(x) \in \textsc{conc}(\sigma(x))$. Because p is valid, $\sigma(x) \sqsubseteq \textsc{safe}[\![\iota]\!](x)$, so $\rho(x) \in \textsc{conc}(\textsc{safe}[\![\iota]\!](x))$. Finally, x was arbitrary, so by the property of the safety function, $\xi \longrightarrow_p \xi'$ for some $\xi' \in \textsc{State}_p$. ◀

▶ **Lemma 12.** *Let $(A, \sqcup)$ be a semilattice (whose join function induces the partial order $\sqsubseteq$), let $\textsc{flow} : \textsc{Inst} \times \textsc{Map}_A \rightharpoonup \textsc{Map}_A$ (where $\textsc{Map}_A = \textsc{Var} \rightharpoonup A$) be monotonic in the second parameter, and let $p \in \textsc{Prog}$. If $\pi = \textsc{Kildall}(\textsc{flow}, \sqcup, p)$ and $[\iota]_{v_1} \xrightarrow{p} v_2$ then $\textsc{flow}[\![\iota]\!](\pi(v_1)) \sqsubseteq \pi(v_2)$.*

Proof. We proceed by showing that the following is a loop invariant for the **while** loop in lines 4–15 of Algorithm 1: if $[\iota]_{v_1} \xrightarrow{p} v_2$ and $\textsc{flow}[\![\iota]\!](\pi(v_1)) \not\sqsubseteq \pi(v_2)$, then $v_1 \in V$. On the first iteration, the invariant clearly holds because $V = \textsc{Vert}_p$. Now, assume that the invariant holds at the beginning of an iteration. We show that the following is a loop invariant for the **for** loop in lines 9–14: if U is the set of all u that we have not reached yet, then all violations of the outer invariant have $v_1 = v$ and $v_2 \in U$. This holds at the first iteration because the only thing we removed from V was v, and π is unchanged. Next assume that the inner invariant holds at the beginning of an iteration of the inner loop. The **if** statement in lines 10–13 runs iff v, u violate the outer invariant. Because $\sigma' \sqsubseteq \pi(u) \sqcup \sigma'$, no violation with $v_1 = v$ has $v_2 = u$ after line 11, although we may now have some violations with $v_1 = u$. But after line 12, we no longer have any violations involving u, so all violations now have $v_2 \in U \setminus \{u\}$ and again $v_1 = v$. After this inner loop exits, we no longer have any violations of the outer invariant because $U = \varnothing$, so the outer invariant also holds. This completes the proof, because $V = \varnothing$ when the outer loop exits. ◀

Proposition 2:

Proof. Let $\pi = \textsc{Kildall}(\textsc{flow}, \sqcup, p)$. Then let $\langle S_1 \parallel \mu_1 \rangle = \xi$ and $\langle S_2 \parallel \mu_2 \rangle = \xi'$. If $S_2 = \langle \varnothing, v_2 \rangle \cdot S_1$ then $\pi(v_2)$ describes $\varnothing$ vacuously. Otherwise, $S_1 = S' \cdot \langle \rho_1, v_1 \rangle \cdot S$ and $S_2 = \langle \rho_2, v_2 \rangle \cdot S$ where $v_1 \xrightarrow{p} v_2$. Let $\sigma_1 = \pi(v_1)$ and $\sigma_2 = \pi(v_2)$. Because ξ is valid, σ_1 describes ρ_1. By local soundness, $\sigma_2' = \textsc{flow}[\![\iota]\!](\sigma_1)$ describes ρ_2. Then $\sigma_2' \sqsubseteq \sigma_2$ by Lemma 12 (with $A = \textsc{Abst}$), so σ_2 describes ρ_2. In each of these cases, the top stack frame of S_2 is valid. All other frames are the same as those of S_1, so ξ' is valid. ◀

▶ **Proposition 13.** *$\widetilde{\textsc{Abst}}$ is the subset of the full lifting generated by $\textsc{Ann}$ via $\widetilde{\sqcup}$.*

Proof. Let $(\widetilde{\textsc{Abst}}', \widetilde{\sqcup})$ be the full lifting of $\textsc{Abst}$ with the corresponding lifted join function, and let

$$\widetilde{\textsc{Abst}} = \textsc{Abst} \cup \{?\} \cup \{a? : a \in \textsc{Abst}\} \subseteq \widetilde{\textsc{Abst}}'$$

be the small lifting. First note that $a \mathbin{\widetilde{\sqcup}} ? = a?$ for all $a \in \textsc{Abst}$, so $\widetilde{\textsc{Abst}}$ is a subset of the set generated by $\textsc{Ann}$ via $\widetilde{\sqcup}$. Then for $\widetilde{a}, \widetilde{b} \in \widetilde{\textsc{Abst}}$,

$$\widetilde{a} \mathbin{\widetilde{\sqcup}} \widetilde{b} = \begin{cases} a \sqcup b & \text{if } \widetilde{a} = a \in \textsc{Abst} \text{ and } \widetilde{b} = b \in \textsc{Abst} \\ a? & \text{if } \widetilde{a} = a \in \textsc{Abst} \text{ and } \widetilde{b} = ? \\ (a \sqcup b)? & \text{if } \widetilde{a} = a \in \textsc{Abst} \text{ and } \widetilde{b} = b? \text{ for some } b \in \textsc{Abst} \\ ? & \text{if } \widetilde{a} = ? \text{ and } \widetilde{b} = ? \\ b? & \text{if } \widetilde{a} = ? \text{ and } \widetilde{b} = b? \text{ for some } b \in \textsc{Abst} \\ (a \sqcup b)? & \text{if } \widetilde{a} = a? \text{ for some } \in \textsc{Abst} \text{ and } \widetilde{b} = b? \text{ for some } \in \textsc{Abst} \\ \widetilde{b} \mathbin{\widetilde{\sqcup}} \widetilde{a} & \text{otherwise} \end{cases}$$

so $\{\widetilde{a} \mathbin{\widetilde{\sqcup}} \widetilde{b} : \widetilde{a}, \widetilde{b} \in \widetilde{\textsc{Abst}}\} \subseteq \widetilde{\textsc{Abst}}$. Thus, $\widetilde{\textsc{Abst}}$ is equal to the set generated by $\textsc{Ann}$ via $\widetilde{\sqcup}$. ◀

▶ **Proposition 14.** *$\widetilde{\textsc{Abst}}$ has an abstraction function α.*

Proof. Let $\widehat{a} \in \mathcal{P}^+(\textsc{Abst})$, and let $A = \{\widetilde{b} \in \widetilde{\textsc{Abst}} \setminus \{?\} : \widehat{a} \subseteq \gamma(\widetilde{b})\}$. If any such $\gamma(\widetilde{b})$ is a singleton then $\alpha(\widehat{a}) = \widetilde{b}$ and we're done. If $A = \varnothing$ then $\alpha(\widehat{a}) = ?$. Now without loss of generality, we assume that each of those $\widetilde{b}$ elements is of the form $b?$ for some $b \in \textsc{Abst}$; that is, there exists an injective "root" map $r : A \to \textsc{Abst}$ given by $r(b?) = b$. Let $A_0 = r(A)$.

Next we inductively define an ascending chain b_k along with a sequence of sets A_k for $k \in \mathbb{N}$; our base case is A_0. Choose $b_k \in A_k$ and let

$$A_{k+1} = \{b \in A_k : b \sqcup b_k \neq b_k\}.$$

If $A_{k+1} = \varnothing$ then we end the chain. Otherwise, choose $b'_k \in A_{k+1}$ and let $b_{k+1} = b_k \sqcup b'_k$. By the construction of A_{k+1}, we know that $b_{k+1} \neq b_k$, so we have continued our ascending chain to be $b_0 \sqsubset \cdots \sqsubset b_k \sqsubset b_{k+1}$ because

$$b_k \sqcup b_{k+1} = b_k \sqcup (b_k \sqcup b'_k) = (b_k \sqcup b_k) \sqcup b'_k = b_k \sqcup b'_k = b_{k+1}.$$

Let h be the height of $\textsc{Abst}$, so we know that our chain has height $n \leq h$. By construction, for every $b \in A_0$ we have $b \sqcup b_k = b_k$ for some $0 \leq k \leq n$, which means that $\gamma(b?) \supseteq \gamma(b_k?)$. Given that $\gamma(b_0?) \supseteq \cdots \supseteq \gamma(b_n?)$, we see that $\gamma(b_n?) = \bigcap \gamma(A)$, so we can define $\alpha(\widehat{a}) = b_n?$. ◀

Proposition 3:

Proof. We have already shown that $\widetilde{\sqcup}$ is commutative and idempotent, so it only remains to show that $\widetilde{\sqcup}$ is associative. But associativity follows immediately from the proof of Proposition 13. ◀

Proposition 4:

Proof. In this proof, we write $\widetilde{a} \sqsubseteq \widetilde{b}$ to mean $\widetilde{a} \,\widetilde{\sqcup}\, \widetilde{b} = \widetilde{b}$ for $\widetilde{a}, \widetilde{b} \in \widetilde{\textsc{Abst}}$, and also write $\widetilde{a} \sqsubset \widetilde{b}$ to mean $\widetilde{a} \sqsubseteq \widetilde{b}$ and $\widetilde{a} \neq \widetilde{b}$. Note that these are not the same as the lifted relation $\widetilde{\sqsubseteq}$, although $\widetilde{\sqsubseteq}$ and this definition of $\sqsubseteq$ both coincide when restricted to $\textsc{Abst} \times \textsc{Abst}$.

By the definition of height, there exists a (not necessarily unique) longest ascending chain $a_0 \sqsubset \cdots \sqsubset a_n$ in $\textsc{Abst}$. Since $n > 0$ we know that $\gamma(a_{n-1}?)$ is not a singleton because $a_{n-1}, a_n \in \gamma(a_{n-1}?)$. Thus, $a_{n-1}? \neq a_{n-1}$. We can then calculate

$$a_{n-1} \,\widetilde{\sqcup}\, a_{n-1}? = (a_{n-1} \sqcup a_{n-1})? = a_{n-1}?,$$
$$a_{n-1}? \,\widetilde{\sqcup}\, a_n? = (a_{n-1} \sqcup a_n)? = a_n?,$$

so $a_{n-1} \sqsubset a_{n-1}? \sqsubset a_n?$ because $a_{n-1} \neq a_n$ implies $a_{n-1}? \neq a_n?$. This shows that the height of the small lifting is at least $n+1$.

Now assume that there exists an ascending chain $\widetilde{a}_0 \sqsubset \cdots \sqsubset \widetilde{a}_{n+2}$ in $\widetilde{\textsc{Abst}}$. Note that for $k > 0$, if $\widetilde{a}_k = ?$ then $\widetilde{a}_{k-1} \,\widetilde{\sqcup}\, ? = ?$, which implies $\widetilde{a}_{k-1} = \bot$, so $\widetilde{a}_k = \bot?$. Thus for $k > 0$ either $\widetilde{a}_k = a_k$ or $\widetilde{a}_k = a_k?$, allowing us to define a new chain $a_1 \sqsubseteq \cdots \sqsubseteq a_{n+2}$. If $\widetilde{a}_0 = ?$ then we must have $\widetilde{a}_1 = a_1? \neq a_1$, because $a_1, a_2 \in \gamma(a_1?)$. In this case we can replace $\widetilde{a}_0$ with a_1, so without loss of generality we can assume that no element of the chain is $?$. Next, if $\widetilde{a}_k = a_k?$ for some $0 \leq k < n+2$, we can use $\widetilde{a}_k \sqsubseteq \widetilde{a}_{k+1}$ to see that $\widetilde{a}_{k+1} = \widetilde{a}_k \,\widetilde{\sqcup}\, \widetilde{a}_{k+1} = a_k? \,\widetilde{\sqcup}\, \widetilde{a}_{k+1} = a_{k+1}?$. By induction this means that if $\widetilde{a}_k = a_k$ and $\widetilde{a}_{k+1} = a_{k+1}?$ for some k, we must have $\widetilde{a}_i = a_i$ for all $i \leq k$ and $\widetilde{a}_j = a_j?$ for all $j > k$. In other words, we have a chain

$$x_0 \sqsubset \cdots \sqsubset x_k \sqsubseteq x_{k+1} \sqsubset \cdots \sqsubset x_{n+2}$$

implying that $\textsc{Abst}$ is at least height $n+1$, contrary to our earlier assumption. Thus the height of the small lifting is at most $n+1$. ◀

Proposition 5:

Proof. For any $a, b \in \textsc{Abst}$ we have $\gamma(a) = \{a\}$ and $\gamma(b) = \{b\}$, so $a \mathbin{\widetilde{\sqcup}} b = \alpha(\{a \sqcup b\}) = a \sqcup b$ because $\gamma(a \sqcup b) = \{a \sqcup b\}$. Thus $\widetilde{\sqcup}$ is a conservative extension of $\sqcup$. Similarly $\widetilde{\textsc{flow}}[\![\iota]\!](\sigma) = \textsc{flow}[\![\iota]\!](\sigma)$ for $\iota \in \textsc{Inst}'$ and $\sigma \in \textsc{Map}$, so $\widetilde{\textsc{flow}}$ is a conservative extension of $\textsc{flow}$. Because $\pi = \textsc{Kildall}(\textsc{flow}, \sqcup, p)$ is well-defined, it follows that $\textsc{Kildall}(\widetilde{\textsc{flow}}, \widetilde{\sqcup}, p) = \pi$. ◀

Proposition 6:

Proof. The predicate $\widetilde{\sqsubseteq}$ is a conservative extension of $\sqsubseteq$, and the function $\widetilde{\textsc{safe}}$ is a conservative extension of $\textsc{safe}$, so p is statically valid according to the gradual analysis as well as valid according to the static analysis. If $\xi_1 \mathrel{\widetilde{\longrightarrow}_p} \xi_2$ then trivially $\xi_1 \longrightarrow_p \xi_2$ because $\xi_2 \neq \texttt{error}$. Conversely, assume that $\xi_1 \longrightarrow_p \xi_2$. Let $\pi = \textsc{Kildall}(\textsc{flow}, \sqcup, p)$. Since p and ξ_1 are valid, if $\xi_1 = \langle\langle \rho, [\iota]\rangle \cdot S \parallel \mu\rangle$ then $\textsc{desc}(\rho, \{x \mapsto \textsc{safe}[\![\iota]\!](x) : x \in \textsc{Var}\})$ by the same reasoning used in the proof of Proposition 1. Then ξ_1 does not step to $\texttt{error}$ because $\widetilde{\textsc{desc}}$ and $\widetilde{\textsc{safe}}$ are conservative extensions of $\textsc{desc}$ and $\textsc{safe}$ respectively. Thus, $\xi_1 \mathrel{\widetilde{\longrightarrow}_p} \xi_2$. ◀

▶ **Lemma 15.** *If $\iota_1, \iota_2 \in \textsc{Inst}$ and $\iota_1 \lesssim \iota_2$, then $\gamma(\widetilde{\textsc{safe}}[\![\iota_1]\!](x)) \subseteq \gamma(\widetilde{\textsc{safe}}[\![\iota_2]\!](x))$ for all $x \in \textsc{Var}$.*

Proof. Let $x \in \textsc{Var}$. If $\widetilde{\textsc{safe}}[\![\iota_1]\!](x) = \widetilde{\textsc{safe}}[\![\iota_2]\!](x)$ then the claim clearly holds. Otherwise, since ι_1 and ι_2 only differ in annotations, there must exist $\iota_1', \iota_2' \in \textsc{Inst}'$ such that $\textsc{safe}[\![\iota_1']\!](x) \neq \textsc{safe}[\![\iota_2']\!](x)$. Therefore we know that $\widetilde{\textsc{safe}}[\![\iota_1]\!](x)$ and $\widetilde{\textsc{safe}}[\![\iota_2]\!](x)$ come from corresponding operands of ι_1 and ι_2 respectively. Since $\iota_1 \lesssim \iota_2$, that operand must be $\widetilde{\textsc{safe}}[\![\iota_2]\!](x) = \texttt{?}$ for ι_2 in order for the safety values to be different. Thus we have $\gamma(\widetilde{\textsc{safe}}[\![\iota_1]\!](x)) \subseteq \gamma(\texttt{?}) = \gamma(\widetilde{\textsc{safe}}[\![\iota_2]\!](x))$. ◀

▶ **Lemma 16.** *Let $p \in \textsc{Prog}$ and $\xi = \langle\langle \rho, [\iota]_v\rangle \cdot E \cdot S \parallel \mu\rangle \in \textsc{State}_p$. If $\widetilde{\textsc{desc}}(\rho, \{x \mapsto \widetilde{\textsc{safe}}[\![\iota]\!](x) : x \in \textsc{Var}\})$ then $\xi \longrightarrow_p \xi'$ for some $\xi' \in \textsc{State}_p$.*

Proof. We know there exists a program $p' \in \textsc{Prog}'$ more precise than p whose states and semantics are the same as those of p, so in particular $\xi \in \textsc{State}_{p'}$, but $\iota' = \textsc{inst}_{p'}(v)$ is not necessarily equal to ι since all instances of $\texttt{?}$ in p have been replaced with $\top$ in p'. Next, by the definition of $\widetilde{\textsc{desc}}$ there exists some $\sigma \in \textsc{Map}$ such that $\textsc{desc}(\rho, \sigma)$ and $\sigma(x) \in \gamma(\widetilde{\textsc{safe}}[\![\iota]\!](x))$ for all $x \in \textsc{Var}$. Now let $x \in \textsc{Var}$. If $\widetilde{\textsc{safe}}[\![\iota]\!](x) = a \in \textsc{Abst}$ then $\textsc{safe}[\![\iota']\!](x) = \sigma(x)$, so $\rho(x) \in \textsc{conc}(a)$. Otherwise there exist $\iota_1, \iota_2 \in \textsc{Inst}'$ such that $\textsc{safe}[\![\iota_1]\!](x) \neq \textsc{safe}[\![\iota_2]\!](x)$, so we know that $\textsc{safe}[\![\iota']\!](x)$ is an operand of ι'. But the corresponding operand of ι must be $\texttt{?}$ since otherwise we would not have multiple values in $\gamma(\widetilde{\textsc{safe}}[\![\iota]\!](x))$, so we have $\textsc{safe}[\![\iota']\!](x) = \top$ and trivially $\rho(x) \in \textsc{conc}(\top)$. Thus, $\textsc{desc}(\rho, \{x \mapsto \textsc{safe}[\![\iota']\!](x) : x \in \textsc{Var}\}))$, so $\xi \longrightarrow_p \xi'$ for some $\xi' \in \textsc{State}_{p'} = \textsc{State}_p$, which means $\xi \longrightarrow_p \xi'$. ◀

Proposition 7:

Proof. Let $\langle \rho, [\iota]\rangle = E_1$. If $\neg\widetilde{\textsc{desc}}(\rho, \{x \mapsto \widetilde{\textsc{safe}}[\![\iota]\!](x) : x \in \textsc{Var}\})$ then $\xi \mathrel{\widetilde{\longrightarrow}_p} \texttt{error}$. Otherwise, $\xi \longrightarrow_p \xi'$ for some $\xi' \in \textsc{State}_p \subset \widetilde{\textsc{State}}_p$ by Lemma 16, so $\xi \mathrel{\widetilde{\longrightarrow}_p} \xi'$ because ξ does not step to $\texttt{error}$. ◀

▶ **Lemma 17.** *Let $p \in \textsc{Prog}$ and $\widetilde{\sigma} \in \widetilde{\textsc{Map}}$, and let $\xi = \langle S' \cdot \langle \rho, [\iota]_v\rangle \cdot S \parallel \mu\rangle$ and $\xi' = \langle\langle \rho', v'\rangle \cdot S \parallel \mu\rangle$. If $\xi \longrightarrow_p \xi'$ and $\widetilde{\textsc{desc}}(\rho, \widetilde{\sigma})$, then $\widetilde{\textsc{desc}}(\rho', \widetilde{\textsc{flow}}[\![\iota]\!](\widetilde{\sigma}))$.*

Proof. We know there exists a program $p' \in \textsc{Prog}'$ more precise than p whose states and semantics are the same as those of p, so in particular $\xi, \xi' \in \textsc{State}_{p'}$, and $\xi \longrightarrow_{p'} \xi'$. However, $\iota' = \textsc{inst}_{p'}(v)$ is not necessarily equal to ι since all instances of $\texttt{?}$ in p have been replaced with $\top$ in p'. Next, by the definition of $\widetilde{\textsc{desc}}$ there exists some $\sigma \in \textsc{Map}$ such that $\textsc{desc}(\rho, \sigma)$ and $\sigma(x) \in \gamma(\widetilde{\sigma}(x))$ for all $x \in \mathrm{dom}(\widetilde{\sigma})$. By local soundness, $\textsc{desc}(\rho', \textsc{flow}[\![\iota']\!](\sigma))$. But by the definition of $\widetilde{\textsc{flow}}$ we know $(\textsc{flow}[\![\iota']\!](\sigma))(x) \in \gamma((\widetilde{\textsc{flow}}[\![\iota]\!](\widetilde{\sigma}))(x))$ for all $x \in \mathrm{dom}(\textsc{flow}[\![\iota']\!](\sigma))$, so $\widetilde{\textsc{desc}}(\rho', \widetilde{\textsc{flow}}[\![\iota]\!](\widetilde{\sigma}))$. ◀

Proposition 8:

Proof. Let $\widetilde{\pi} = \textsc{Kildall}(\widetilde{\textsc{flow}}, \widetilde{\sqcup}, p)$. Then let $\langle S_1 \parallel \mu_1\rangle = \xi$ and $\langle S_2 \parallel \mu_2\rangle = \xi'$. If $S_2 = \langle \varnothing, v_2\rangle \cdot S_1$ then $\widetilde{\pi}(v_2)$ describes $\varnothing$ vacuously. Otherwise, $S_1 = S' \cdot \langle \rho_1, v_1\rangle \cdot S$ and $S_2 = \langle \rho_2, v_2\rangle \cdot S$ where $v_1 \xrightarrow{p} v_2$. Let $\widetilde{\sigma}_1 = \widetilde{\pi}(v_1)$ and $\widetilde{\sigma}_2 = \widetilde{\pi}(v_2)$. Because ξ is valid, $\widetilde{\sigma}_1$ describes ρ_1. By Lemma 17, $\widetilde{\sigma}'_2 = \widetilde{\textsc{flow}}[\![\iota]\!](\widetilde{\sigma}_1)$ describes ρ_2. Then $\widetilde{\sigma}'_2 \widetilde{\sqcup} \widetilde{\sigma}_2 = \widetilde{\sigma}_2$ by Lemma 12 (with $A = \widetilde{\textsc{Abst}}$, $\sqcup = \widetilde{\sqcup}$, and $\textsc{flow} = \widetilde{\textsc{flow}}$), so $\widetilde{\sigma}_2$ describes ρ_2. In each of these cases, the top stack frame of S_2 is valid. All other frames are the same as those of S_1, so ξ' is valid. ◀

▶ **Lemma 18.** *Let $\iota_1, \iota_2 \in \textsc{Inst}$ such that $\iota_1 \lesssim \iota_2$. Then $\gamma((\widetilde{\textsc{flow}}[\![\iota_1]\!](\widetilde{\sigma}))(x)) \subseteq \gamma((\widetilde{\textsc{flow}}[\![\iota_2]\!](\widetilde{\sigma}))(x))$ for all $\widetilde{\sigma} \in \widetilde{\textsc{Map}}$ and $x \in \textsc{Var}$.*

Proof. Using the notation from the definition of $\widetilde{\textsc{flow}}$, we have $I_1 \subseteq I_2$, so the lemma holds by the properties of α. ◀

▶ **Lemma 19.** *Let $p_1, p_2 \in \textsc{Prog}$ such that $p_1 \lesssim p_2$. Let $\pi_1 = \textsc{Kildall}(\widetilde{\textsc{flow}}, \widetilde{\sqcup}, p_1)$ and $\pi_2 = \textsc{Kildall}(\widetilde{\textsc{flow}}, \widetilde{\sqcup}, p_2)$. Let $v \in \textsc{Vert}_{p_1} = \textsc{Vert}_{p_2}$. Let $\sigma_1 = \pi_1(v)$ and $\sigma_2 = \pi_2(v)$. Then $\gamma(\sigma_1(x)) \subseteq \gamma(\sigma_2(x))$ for all $x \in \mathrm{dom}(\sigma_1)$.*

Proof. We proceed by running Algorithm 1 in parallel for p_1 and p_2 and showing that the lemma statement is a loop invariant for the **while** loop in lines 4–15. On the first iteration, the invariant clearly holds because $\mathrm{dom}(\widetilde{\sigma}_1) = \varnothing$. Now, assume that the invariant holds at the beginning of an iteration. Without loss of generality we can assume v to be chosen to be the same for both sides, because if $v_1 \notin V_2$ or $v_2 \notin V_1$ then the **if** statement on line 10 will never run for the first or second side, respectively. After line 7 we have $\gamma(\sigma_1(x)) \subseteq \gamma(\sigma_2(x))$ for all x by assumption. Then after line 8 we have $\gamma(\sigma'_1(x)) \subseteq \gamma(\sigma'_2(x))$ for all x by Lemma 18. The in the inner **for** loop, we enter the **if** statement in line 10 exactly when the assignment statement on line 11 would have an effect. By the properties of $\widetilde{\sqcup}$, the invariant still holds for $\pi_1(u)$ and $\pi_2(u)$ after line 11. This accounts for all the elements of π_1 and π_2 that we change. We have thus completed the proof. ◀

Proposition 9:

Proof.
Let $\widetilde{\pi}_1 = \textsc{Kildall}(\widetilde{\textsc{flow}}, \widetilde{\sqcup}, p_1)$ and $\widetilde{\pi}_2 = \textsc{Kildall}(\widetilde{\textsc{flow}}, \widetilde{\sqcup}, p_2)$. Let $v \in \textsc{Vert}_{p_1} = \textsc{Vert}_{p_2}$ and $x \in \textsc{Var}$, let $\iota_1 = \textsc{inst}_{p_1}(v)$ and $\iota_2 = \textsc{inst}_{p_2}(v)$, and let $\widetilde{\sigma}_1 = \widetilde{\pi}_1(v)$ and $\widetilde{\sigma}_2 = \widetilde{\pi}_2(v)$. By Lemma 19 we know $\gamma(\widetilde{\sigma}_1(x)) \subseteq \gamma(\widetilde{\sigma}_2(x))$. Also, by Lemma 15 we know $\gamma(\widetilde{\textsc{safe}}[\![\iota_1]\!](x)) \subseteq \gamma(\widetilde{\textsc{safe}}[\![\iota_2]\!](x))$. Then by the definition of $\widetilde{\sqsubseteq}$, if $\widetilde{\sigma}_1(x) \mathrel{\widetilde{\sqsubseteq}} \widetilde{\textsc{safe}}[\![\iota_1]\!](x)$ then $\widetilde{\sigma}_2(x) \mathrel{\widetilde{\sqsubseteq}} \widetilde{\textsc{safe}}[\![\iota_2]\!](x)$. ◀

Proposition 10:

Proof. Because $\xi_2 \neq \texttt{error}$, we know that $\xi_1 \longrightarrow_{p_1} \xi_2$. This means that $\xi_1 \longrightarrow_{p_2} \xi_2$ because ξ_2 is the same as ξ_1 except with possibly some annotations removed. Thus, it only remains to show that ξ_1 does not step to $\texttt{error}$ under $\widetilde{\longrightarrow}_{p_2}$. Assume that $\xi = \langle\langle \rho, v\rangle \cdot S \parallel \mu\rangle$ where $\textsc{inst}_{p_1}(v) = \iota_1$ and $\textsc{inst}_{p_2}(v) = \iota_2$. Because ξ_1 does not step to $\texttt{error}$, we know that $\widetilde{\textsc{desc}}(\rho, \{x \mapsto \widetilde{\textsc{safe}}[\![\iota_1]\!](x) : x \in \textsc{Var}\})$. This means that there exists some $\sigma \in \textsc{Map}$ such that $\text{DESC}(\rho, \sigma)$ and $\sigma(x) \in \gamma(\widetilde{\text{SAFE}}[\![\iota_1]\!](x))$ for all $x \in \text{VAR}$. By Lemma 15 we know that $\sigma(x) \in \gamma(\widetilde{\text{SAFE}}[\![\iota_1]\!](x))$ for all $x \in \text{VAR}$. This completes the proof, because by the definition of DESC we now know that $\widetilde{\text{DESC}}(\rho, \{x \mapsto \widetilde{\text{SAFE}}[\![\iota_2]\!](x) : x \in \text{VAR}\})$, so ξ_1 does not step to `error` under $\widetilde{\longrightarrow}_{p_2}$, so $\xi_1 \widetilde{\longrightarrow}_{p_2} \xi_2$. ◀

Proposition 11:

Proof. We know that $\neg\widetilde{\text{DESC}}(\rho, \{x \mapsto \widetilde{\text{SAFE}}[\![\iota]\!](x) : x \in \text{VAR}\})$. By the definitions of $\widetilde{\text{DESC}}$ and DESC, there is some $x \in \text{VAR}$ and $b \in \gamma(\widetilde{\text{SAFE}}[\![\iota]\!](x))$ such that $\rho(x) \notin \text{CONC}(b)$. But since ξ is valid, there exists some $a \in \gamma((\widetilde{\pi}(v))(x))$ such that $\rho(x) \in \text{CONC}(a)$. Thus, $\text{CONC}(a) \nsubseteq \text{CONC}(b)$, so $a \not\sqsubseteq b \sqsubseteq \bigsqcup \gamma(\widetilde{\text{SAFE}}[\![\iota]\!](x))$. ◀